LynxOS Reference Manual

LynxOS Version 2.5
DOC-0172

Library Functions (A–M)

U.S. Patents 5,469,571; 5,594,903

Printed in the United States of America.

Contents

Library Functions (A-M)

Library Functions (A-M)

NAME

`Intro` – LynxOS software support library

DESCRIPTION

The Library Functions Reference Manual spans two volumes in its printed form: “Library Functions (A–M)” and “Library Functions (N–Z)”. A master index is provided in the back of the second volume, “Library Functions (N–Z)”.

The LynxOS software support library contains routines which perform a wide variety of common operations for LynxOS applications programs. These routines are designed to be as efficient as possible. Hopefully, they will not need to be re-invented by each application designer.

In many cases, library routines, which return data types other than `int`, are appropriately declared in some header file which is included when using the routines. Because this is not always the case, however, care should be taken to ensure the routines are declared.

NAME

`a64l, l64a` – convert between base-64 ASCII string and long integer

SYNOPSIS

```
long a64l(s)
char *s;

char *l64a(l)
long l;
```

DESCRIPTION

`a64l` converts the base-64 ASCII string pointed to by **`s`** to a **`long`** value. The string **`s`** is scanned up to the first unrecognized character or until six characters have been scanned, whichever comes first. Unrecognized characters include the null character or any character not used in the base-64 notation described below.

The characters used to represent “digits” in base-64 notation are:
. for 0; / for 1; 0 through 9 for 2-11; A through Z for 12-37; and a through z for 38-63.

`l64a` converts the **`long`** argument **`l`** to a base-64 ASCII string. If **`l`** is 0, `l64a` returns a pointer to the null string. The value returned by `l64a` is a pointer into a static buffer, the contents of which are overwritten by each call.

The base-64 ASCII strings used by these routines are interpreted as having their most significant digit to the right, not to the left. Thus, the base-64 ASCII string “AZ” converts to 2380, not 805.

NAME

`abort` – abort process

SYNOPSIS

`void abort()`

DESCRIPTION

`abort` sends the signal `SIGABRT` to the calling process. If `SIGABRT` is neither being caught nor ignored, all `stdio` streams are flushed prior to the signal being sent, and a core dump results.

DIAGNOSTICS

`abort` causes abornal program termination (core dumps) to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return.

SEE ALSO

Library Functions - `exit(), signal()`

System Calls - `kill(), sigaction()`

NAME

`abs, labs` – integer or long absolute value

SYNOPSIS

```
int abs(i)
int i;

long labs(l)
long l;
```

DESCRIPTION

`abs` (`labs`) returns the absolute value of its **`int`** (**`long`**) operand.

SEE ALSO

Library Functions - `fabs()`

NAME

afsync – asynchronous file synchronization

SYNOPSIS

```
#include <fcntl.h>
#include <sys/aio.h>

int afsync(fd, op, aiocbp)
int fd;
int op;
struct aiocb *aiocbp;
```

DESCRIPTION

The afsync call ensures that any memory-resident data of the file open with the descriptor **fd** is written to the device on which the file resides. The afsync call is asynchronous, returning when the request has been queued. The sync operation is thus performed by a separate thread of execution, in parallel with computation by the calling thread. The afsync call will return to the calling thread before completion of the actual sync operation. Thus, while the afsync call is asynchronously doing the sync operation, the calling thread continues with the next instruction.

The **op** argument must be one of:

O_DSYNC — All I/O operations are completed to maintain DATA integrity.

O_FSYNC — All I/O operations are completed to maintain both DATA and FILE integrity. File integrity refers to attributes associated with the file such as size and mode.

The **aiocbp** argument points to a asynchronous I/O control block.

The structure **aiocb** is defined in <sys/aio.h>. The relevant fields are as follows (not necessarily in this order):

```
off_t          aio_offset;
int            aio_whence;
int            aio_flag;
struct event   aio_event;
```

```
int             aio_errno;
size_t          aio_nobytes;
int             aio_prio;       /* not currently used  */
```

The events structure, defined in `<sys/events.h>` contains the following fields (not necessarily in this order):

```
evt_class_t  evt_class;
void         *evt_value;
void         (*evt_handler)() ;  /* not used by LynxOS */
evtset_t     evt_classmask ;     /* not used by LynxOS */
```

Looking ahead to POSIX draft 10, Lynx only uses the `evt_value` and `evt_class` fields of the event structure. For the same reason, we use the words “event” and “signal” interchangeably in this document, and use the `sigaction` system call to set up the action to be taken when the event is received by the calling process.

The following is a sample event handler:

```
void
aio_write_handler(signo, value)
int  signo;
void *value;
{
  printf("in aio_write_handler:  ");
  printf("signo = %d, value = %ld\n", signo, (long) value);
}
```

If `aio_flag` contains the flag `AIO_EVENT`, then, when the asynchronous operation completes, the signal (`event`) `aio_event.evt_class` is sent to the calling thread, with event data `aio_event.evt_value`. If `aio_event.evt_class` is 0, no signal is sent. The calling thread can set the action to be performed on receiving the signal to an appropriate event handler (similar to a signal handler). The calling thread should use the `sigaction` system call to set up the action to be performed on receiving the signal.

If `aio_flag` is 0, then no signal will be sent on completion of the operation.

When the request is first submitted, `EINPROG` is written to `aio_errno`; this value may be polled to determine when the asynchronous operation completes. If the I/O operation completes successfully, the `aio_errno` is set to 0, and `aio_nobytes` is set to the return value of the `sync` operation. If the I/O operation fails, `aio_errno` is set to the appropriate error code, and `aio_nobytes` is set to -1.

In POSIX Draft 10, the return of the operation's return value and error code will be handled more gracefully. `aio_nobytes` is currently the field of choice due to the actions of `aread` and `awrite`.

The `aio_prio` field is currently ignored. If the `afsync` call fails, the `sync` operation is not guaranteed to have completed successfully.

If **aiocbp** is NULL then no status is returned and no event notification is given on completion of the operation. In this case, if the operation fails, `aio_nobytes` and `aio_errno` cannot be set. Note that there is thus no way to know that the `afsync` call has completed. We recommend against using a value of NULL for the **aiocbp**.

To use this function, include the library `libc_p.a` in the compilation command line.

DIAGNOSTICS

If `afsync` succeeds, 0 is returned; if it fails, -1 is returned and **errno** is set to one of the following values.

EAGAIN	All the asynchronous I/O threads are currently busy; no more memory exists for creating new events; no more asynchronous I/O buffers exit for queueing the operation.
EBADF	The **fd** argument is not a valid file descriptor open for writing.
EINVAL	A value of **op** other than O_DSYNC or O_FSYNC was specified.
EFTYPE	**fd** does not reference a regular file.
EFAULT	**aiocbp** is not a valid address.

If the `afsync` call returns successfully, but later fails, `aio_errno` is set to `EINVAL`. Note that if the call fails asynchronously, it will still send a signal to the calling process.

`EINVAL`	The file is not a regular file.

CONFORMANCE

`afsync` is implemented according to Draft 9 of the Posits 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `fsync()`

Library Functions - `aread(), awrite(), listio()`

The POSIX asynchronous I/O calls are:

System Calls - `acancel()`

Library Functions - `iosuspend(), aread(), awrite(), listio(), afsync().`

Example - `afsync.c`

NAME

aio_cancel – cancel asynchronous I/O request

SYNOPSIS

```
#include <aio.h>

int aio_cancel(fildes, aiocbp)
int fildes;
struct aiocb *aiocbp;
```

DESCRIPTION

aio_cancel attempts to cancel one or more asynchronous I/O operations that are pending for the file descriptor **fildes**. If **aiocbp** is NULL, then all outstanding asynchronous I/O requests pending for **fildes** are canceled. If aiocbp is not NULL, then aio_cancel cancels a specific asynchronous I/O request for **fildes**, specified by **aiocbp**. For all asynchronous I/O operations that are canceled, a normal signal delivery will occur based on the aiocb structures for those operations. Also, the error status for each canceled operation will be set to ECANCELED, and the return status for each canceled operation will be set to –1. Those asynchronous I/O operations that are not canceled will complete normally.

DIAGNOSTICS

aio_cancel cancels the requested asynchronous I/O operations when successful, and returns a value of AIO_CANCELED. If at least one of the requested operations cannot be canceled, it returns a value of AIO_NOTCANCELED. If all requested asynchronous I/O operations had already completed before aio_cancel could cancel them, it returns a value of AIO_ALLDONE. Otherwise, aio_cancel returns a value of –1 and sets **errno** as follows:

EBADF	**fildes** is not a valid file descriptor.

CONFORMANCE

aio_cancel is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `aio_read(), aio_write(), lio_listio(), aio_error(), aio_return()`

NAME

aio_error – retrieve error status of asynchronous I/O operation

SYNOPSIS

```
#include <aio.h>

int aio_error(aiocbp)
const struct aiocb *aiocbp;
```

DESCRIPTION

aio_error returns the error status associated with an asynchronous I/O operation. The only argument to this function, **aiocbp**, is a pointer to the aiocb structure which describes the asynchronous I/O operation. This structure is defined in **<aio.h>** and described in the aio_read manual page. The error status for an asynchronous I/O operation is the **errno** value that would have been set for the corresponding read, write, or fsync operation. If the operation has not yet completed, the error status is EINPROGRESS.

DIAGNOSTICS

aio_error returns 0 if the asynchronous I/O operation has completed successfully. If the operation has completed unsuccessfully, it returns the **errno** for a corresponding read, write, or fsync operation. If the operation has not yet completed, it returns a value of EINPROGRESS. Otherwise, **aio_error** returns –1 and sets **errno** as:

EINVAL	The **aiocbp** argument does not correspond to a valid asynchronous I/O operation for which the return status has not yet been retrieved.
ENOSYS	The aio_error function is not supported.

CONFORMANCE

aio_error is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `aio_read(), aio_write(), lio_listio(), aio_fsync(), aio_return()`

NAME

aio_fsync – asynchronous file synchronization

SYNOPSIS

```
#include <aio.h>

int aio_fsync(op, aiocbp)
int op;
struct aiocb *aiocbp;
```

DESCRIPTION

aio_fsync asynchronously forces I/O operations associated with a file to synchronization. The **aiocbp** argument specifies a pointer to an **aiocb** structure, which is defined in **<aio.h>** as follows:

Type	Name	Description
int	aio_fildes	File descriptor
off_t	aio_offset	File offset (N/A)
volatile void *	aio_buf	Pointer to buffer (N/A)
size_t	aio_nbytes	Length of data (N/A)
int	aio_reqprio	Async I/O priority
struct sigevent	aio_sigevent	Signal number and value
int	aio_lio_opcode	List I/O operation (N/A)

The file descriptor aiocbp->aio_fildes specifies the file of interest. aio_fsync causes all queued I/O operations for this file of a priority equal to or higher than aiocbp->aio_reqprio to be forced to synchronized I/O completion state. The **op** argument specifies the operation according to which the synchronization is performed. It can have two valid values as follows:

O_DSYNC All I/O operations are completed as if by a call to fdatasync; that is synchronized I/O data integrity completion. All read and write operations currently queued to the file descriptor are ensured to be complete.

O_SYNC — All I/O operations are completed as if by a call to fsync; that is synchronized I/O file integrity completion. All read and write operations currently queued to the file descriptor are ensured to be complete. In addition, all file attributes relative to the I/O operation (access time, modification time, status change time) are successfully transferred.

The aiocb structure can be used as an argument in subsequent aio_error and aio_return calls to determine the error status and the return status, respectively, of the asynchronous operation. While the request is queued, the error status for the operation is EINPROGRESS. When the I/O transfer is complete, the error status is updated to reflect the success or failure of the operation.

When the I/O transfer operation is complete, signal notification can be requested. The aiocbp->aio_sigevent member defines the signal to be generated upon completion of the I/O. If the value of aiocbp->aio_sigevent. sigev_signo is non-zero, this signal is generated when all operations have achieved synchronized I/O completion.

A signal handler or a real-time signal handler should be set to handle this signal. Please refer to the aio_read man page for details on setting up a real time signal handler. The aio_offset, aio_buf, aio_nbytes, and aio_lio_opcode members of the **aiocb** structure are ignored by the aio_fsync function. aio_reqprio is not ignored—all requests of greater or equal priority will be completed; all lower priority requests will not be completed.

DIAGNOSTICS

aio_fsync queues the specified I/O operations when successful and returns 0. On failure, it returns -1 and sets **errno** as:

EAGAIN — Due to resource limitations in the system, the requested asynchronous operation was not queued.

EBADF — The aio_fildes member of the **aiocbp** argument is not a valid file descriptor open for writing.

`EINVAL`	An invalid value was specified for **op**. Synchronized I/O is not supported for this file.

If any of the queued I/O operations for the specified file fail, `aio_fsync` returns the error condition for read and write. The error status can be retrieved with the `aio_error` function.

CONFORMANCE

`aio_fsync` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `fsync(), fdatasync(), read(), write()`

Library Calls - `aio_return(), aio_error(), aio_read()`

NAME

`aio_read` – asynchronous read

SYNOPSIS

```
#include <aio.h>

int aio_read(aiocbp)
struct aiocb *aiocbp;
```

DESCRIPTION

`aio_read` asynchronously reads data from a file into a buffer. The only argument to this function is a pointer to an **`aiocb`** structure. This structure is defined in **`<aio.h>`** as:

Type	Name	Description
`int`	`aio_fildes`	File descriptor
`off_t`	`aio_offset`	File offset
`volatile void *`	`aio_buf`	Pointer to buffer
`size_t`	`aio_nbytes`	Length of data
`int`	`aio_reqprio`	Async I/O priority
`struct sigevent`	`aio_sigevent`	Signal number and value
`int`	`aio_lio_opcode`	List I/O operation

The `aio_fildes` element is the file descriptor from which the data will be read asynchronously. The read operation takes place in the file as if `lseek` were called immediately prior to the send with an `offset` argument of `aio_offset`, and a `whence` argument of `SEEK_SET`. The `aio_buf` element specifies to the buffer into which the data will be read.

The `aio_nbytes` element specifies the length of the data, in bytes, to be read.

The aio_reqprio element specifies the asynchronous read priority. The asynchronous read operation is submitted at a priority equal to the calling process's scheduling priority minus aiocbp->aio_reqprio. The error status and the return status of the asynchronous operation can be determined with the aio_error and aio_return functions, respectively, with **aiocbp** as an argument.

The aio_sigevent element specifies how the calling process will be notified upon I/O completion. The sigevent structure is defined in <signal.h> as:

Type	Name	Description
int	sigev_signo	Signal number
union sigval	sigev_value	Signal value
int	sigev_notify	Notification type

If the aio_sigevent.sigev_notify element is SIGEV_NONE, then no signal will be sent upon completion of the I/O. If the aio_sigevent.sigev_notify element is SIGEV_SIGNAL, then the signal specified by aio_sigevent.sigev_signo will be sent to the calling process. If the signal number is a real-time signal (i.e., is in the range SIGRTMIN to SIGRTMAX), and if the SA_SIGINFO flag is set for that signal number with a sigaction call, then the signal will be queued to the calling process. In this case, the aio_sigevent.sigev_value element specifies the value that will be sent along with the real-time signal. A real-time signal handler must be set, for a real-time signal to be queued to the calling process. A real-time signal handler is set by setting the sa_sigaction member of the sigaction structure used in the sigaction call. A real-time signal handler is declared as follows:

```
void signal_handler(signo, info, context)
int signo;
siginfo_t *info
void *context;
```

The **signo** argument is the signal number. The **info** argument points to a siginfo_t structure that encodes the cause of the signal generation, the signal number, and the signal value. The **context** argument is not used by LynxOS. Refer to the sigqueue manual page for more information.

The `aio_lio_opcode` element is ignored by the `aio_read` function.

The `aio_read` function call returns when the asynchronous read request has been initiated or queued.

DIAGNOSTICS

`aio_read` queues the asynchronous read request when successful, and returns a value of `0`. On failure, it returns `-1` and sets **`errno`** as follows:

`EAGAIN`	The requested asynchronous read operation was not queued due to system resource limitations.
`EBADF`	`aiocbp->aio_fildes` is not a valid file descriptor.
`EINVAL`	The file offset of `aiocbp->aio_offset` is invalid, or `aiocbp->aio_reqprio` is not a valid value, or `aiocbp->aio_nbytes` is an invalid value.

If the `aio_read` function successfully queues the read operation, but the operation is later canceled or encounters an error, the return status of the read operation is same as one of the values normally returned by the `read` call. Also, the error status is the same as normally set by the `read` call, or one of the following values:

`EBADF`	`aiocbp->aio_fildes` is not a valid file descriptor open for reading.
`ECANCELED`	The requested I/O was canceled with `aio_cancel` before it could be completed.
`EINVAL`	The file offset `aiocbp->aio_offset` is invalid.

CONFORMANCE

`aio_read` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `read(), lseek(), sigqueue()`

Library Calls - `aio_cancel(), aio_write(), lio_listio(), aio_fsync(), aio_error(), aio_return()`

NAME

aio_return – retrieve return status of asynchronous I/O operation

SYNOPSIS

```
#include <aio.h>

ssize_t aio_return(aiocbp)
struct aiocb *aiocbp;
```

DESCRIPTION

aio_return returns the return status associated with an asynchronous I/O operation. The **aiocbp** argument is a pointer to an **aiocb** structure defined in **<aio.h>** and described in the aio_read manual page. The return status from an asynchronous I/O operation is the value which would be returned by the corresponding read, write, or fsync operation.

aio_return can be called once, and only once, for a given asynchronous I/O operation.

DIAGNOSTICS

aio_return returns the return status for a successfully completed asynchronous I/O operation, as the return value of the corresponding read, write, or fsync function. On failure, it returns –1 and sets **errno** as follows:

EINVAL	The **aiocbp** argument does not correspond to a valid asynchronous I/O operation, whose return status has not yet been retrieved.

CONFORMANCE

aio_return is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls -	aio_read(), aio_write(), lio_listio(), aio_fsync(), aio_error()

NAME

aio_suspend – wait for asynchronous I/O request

SYNOPSIS

```
#include <aio.h>

int aio_suspend(list, nent, timeout)
const struct aiocb *list[];
int nent;
const struct timespec *timeout;
```

DESCRIPTION

aio_suspend is used to suspend the calling process until at least one of the asynchronous I/O requests from **list** is completed, a signal interrupts the function, or if **timeout** is not NULL, the time specified by **timeout** expires, or one of the specified asynchronous I/O request is completed. The argument **list** is an array of pointers to **aiocb** structures. Each **aiocb** structure corresponds to an asynchronous I/O request. The argument **nent** is the number of entries in **list**. The argument **timeout** is used to specify a timed wait for the calling process. If the **timeout** argument is NULL, there is no time constraint on the process suspension.

Each of the asynchronous I/O operations in **list** may have a signal associated with it, when it completes. This signal may interrupt the aio_suspend function, and is indicated as a failure.

The calling process is not suspended if at least one of the asynchronous I/O requests has already completed. In this case, the function returns normally.

Each **aiocb** structure referenced by an entry in **list** must have been used in submitting asynchronous I/O requests with aio_read, aio_write, aio_fsync, or lio_listio functions. The list array may contain NULL pointers, which are ignored.

If the time interval referenced by the **timeout** argument elapses before any of the asynchronous I/O requests are completed, then aio_suspend returns with an error.

DIAGNOSTICS

aio_suspend suspends the calling process and returns after one or more asynchronous I/O operations has completed, and returns 0. On failure it returns -1 and sets **errno** as follows:

EAGAIN	The time interval specified by timeout elapsed before any of the requested asynchronous I/O operations completed.
EINTR	A signal interrupted the aio_suspend call. This signal may be caused by the completion of one of the asynchronous I/O requests.

CONFORMANCE

aio_suspend is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - aio_read(), aio_write(), lio_listio()

NAME

aio_write – asynchronous write

SYNOPSIS

```
#include <aio.h>

int aio_write(aiocbp)
struct aiocb *aiocbp;
```

DESCRIPTION

aio_write to asynchronously writes data to a file from a buffer. The only argument to this function is a pointer to an **aiocb** structure. This structure is defined in **<aio.h>** as:

Type	Name	Description
int	aio_fildes	File descriptor
off_t	aio_offset	File offset
volatile void *	aio_buf	Pointer to buffer
size_t	aio_nbytes	Length of data
int	aio_reqprio	Async I/O priority
struct sigevent	aio_sigevent	Signal number and value
int	aio_lio_opcode	List I/O operation

The aio_fildes element is the file descriptor to which the data will be written asynchronously.

If the O_APPEND flag is set for the file descriptor, then the write operations append to the file in the same order as the calls were made. If the O_APPEND flag is not set for the file descriptor, then the asynchronous write operation takes place in the file as if lseek were called immediately prior to the write with an offset argument of aio_offset, and a whence argument of SEEK_SET.

The aio_buf element specifies the pointer to the buffer which holds the data to be written.

The aio_nbytes element specifies the length of the data in bytes, to be written. The aio_reqprio element specifies the asynchronous write priority. The asynchronous write operation is submitted at a priority equal to the calling process's scheduling priority minus aiocbp->aio_reqprio. The error status and the return status of the asynchronous operation can be determined with the aio_error and aio_return functions respectively, with **aiocbp** as an argument.

The aio_sigevent element specifies how the calling process will be notified upon I/O completion. The sigevent structure is defined in <signal.h> as:

Type	Name	Description
int	sigev_signo	Signal number
union sigval	sigev_value	Signal value
int	sigev_notify	Notification type

If the aio_sigevent.sigev_notify element is SIGEV_NONE, then no signal will be sent upon completion of the I/O. If the aio_sigevent.sigev_notify element is SIGEV_SIGNAL, then the signal specified by aio_sigevent.sigev_signo will be sent to the calling process. If the signal number is a real-time signal (i.e., is in the range SIGRTMIN to SIGRTMAX), and if the SA_SIGINFO flag is set for that signal number with a sigaction call, then the signal will be queued to the calling process. In this case, the aio_sigevent.sigev_value element specifies the value that will be sent along with the real-time signal.

A real-time signal handler must be set, for a real-time signal to be queued to the calling process. A real-time signal handler is set by setting the sa_sigaction member of the sigaction structure used in the sigaction call. A real-time signal handler is entered as follows:

```
void signal_handler(signo, info, context)
int signo;
siginfo_t *info
void *context;
```

The `signo` argument is the signal number. The `info` argument points to a `siginfo_t` structure that indicates the cause of the signal generation, the signal number, and the signal value. The `context` argument is not used by LynxOS. Refer to the `sigqueue` manual page for more information.

The `aio_lio_opcode` element is ignored by the `aio_write` function.

The `aio_write` function call returns when the asynchronous write request has been queued.

DIAGNOSTICS

`aio_write` queues the asynchronous write request when successful, and returns a value of `0`. On failure, it returns `-1` and sets **`errno`** as follows:

`EAGAIN`	The requested asynchronous write operation was not queued due to system resource limitations.
`EBADF`	`aiocbp->aio_fildes` is not a valid file descriptor.
`EINVAL`	The file offset of `aiocbp->aio_offset` is invalid, or `aiocbp->aio_reqprio` is not a valid value, or `aiocbp->aio_nbytes` is an invalid value.

If the `aio_write` function successfully queues the write operation, but the operation is later canceled or encounters an error, the return status of the write operation is same as one of the values normally returned by the `write` call. Also, the error status is the same as normally set by the `write` call, or one of the following values:

`EBADF`	`aiocbp->aio_fildes` is not a valid file descriptor open for writing.
`ECANCELED`	The requested I/O was canceled with `aio_cancel` before it could be completed.
`EINVAL`	The file offset `aiocbp->aio_offset` is invalid.

CONFORMANCE

`aio_write` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `write(), lseek(), sigqueue()`

Library Calls - `aio_cancel(), aio_read(), lio_listio(), aio_fsync(), aio_error(), aio_return()`

NAME

`alarm` – set software timer

SYNOPSIS

```
#include &lt;types.h&gt;

unsigned int alarm(seconds)
unsigned int seconds;
```

DESCRIPTION

`alarm` uses the `setitimer` system call to arrange for a signal (`SIGALRM`) to be sent to the calling process after the given number of seconds have elapsed. Due to inherent characteristics of multi-tasking operating systems, the actual time from `alarm` call to signal receipt may be somewhat greater than that requested. Because only one real-time timer is provided by the operating system, successive calls to `alarm` remove previous timer information; the number of seconds remaining in a previously running timer is returned. A value of zero for **`seconds`** cancels the timer.

The default action for `SIGALRM` is process termination.

SEE ALSO

System Calls - `getitimer()`, `setitimer()`, `sigvec()`

Library Functions - `signal()`

NAME

`alloca` – stack memory allocation

SYNOPSIS

```
char *alloca(size)
unsigned long size;
```

DESCRIPTION

`alloca` allocates **`size`** bytes of memory from the stack of the calling process. The pointer returned is adjusted to a longword boundary to prevent odd address faults. Memory allocated with `alloca` is implicitly freed when the function which allocated the memory returns.

NOTE

Severely bad things happen if memory allocated with `alloca` is given to `free` for deallocation.

SEE ALSO

System Calls - `sbrk()`

Library Functions - `malloc(), free()`

NAME

`aread` – asynchronously read from a file

SYNOPSIS

```
#include <sys/aio.h>

aread(fd, buf, nbytes, aiocbp)
int fd;
char *buf;
size_t nbytes;
struct aiocb *aiocbp;
```

DESCRIPTION

`aread` performs an asynchronous read on the file **fd**, reading **nbytes** into buffer **buf** as if by the corresponding `read` system call.

This read, however, is performed by a separate thread of execution, in parallel with computation by the calling thread. The `aread` call will return to the calling thread before completion of the actual `read` operation. Thus, while the `aread` call is asynchronously doing the `read` operation the calling thread continues with the next instruction. At the completion of the operation, the file pointer will be updated.

The structure **aiocb** is defined in `<sys/aio.h>`. The relevant fields are as follows (not necessarily in this order):

```
off_t           aio_offset;
int             aio_whence;
int             aio_flag;
struct event    aio_event;
int             aio_errno;
size_t          aio_nobytes;
int             aio_prio;  /* not currently used */
```

The events structure, defined in `<sys/events.h>` contains the following fields (not necessarily in this order):

```
evt_class_t    evt_class;
void           *evt_value;
void           (*evt_handler)(); /* not used by LynxOS */
evtset_t       evt_classmask;   /* not used by LynxOS */
```

Looking ahead to POSIX Draft 10, Lynx only uses the `evt_value` and `evt_class` fields of the event structure. For the same reason, we use the words “event” and “signal” interchangeably in this document, and use the `sigaction` system call to set up the action to be taken when the event is received by the calling process.

The event handler for Draft 9 is different from the handler for Draft 10. To request the Draft 9 handler, set the value of the `sigaction sa_flags` to `SA_D9EV`. To request the Draft 10 handler, set the value of the `sigaction sa_flags` to 0.

The following is a sample Draft 10 event handler:

```
void
aio_read_handler(signo, value)
int  signo;
void *value;
{
  printf("in aio_read_handler:  ");
  printf("signo = %d, value = %d\n", signo, value);
}
```

The following is a sample Draft 9 event handler:

```
void
aio_read_handler(event_value, event_class, evt_mask)
void *event_value;
evt_class_t event_class;
evtset_t evt_mask;
{
  printf("in aio_read_handler\n");
}
```

Before the data is read, an lseek is performed on the file **fd** to advance the file pointer to the desired position, using aio_whence and aio_offset as the arguments to lseek.

If aio_flag contains the flag AIO_EVENT, then, when the asynchronous operation completes, the signal (event) aio_event.evt_class is sent to the calling thread, with event data aio_event.evt_value. If aio_event.evt_class is 0, no signal is sent. The calling thread can set the action to be performed on receiving the signal to an appropriate event handler (similar to a signal handler). The calling thread should use the sigaction system call to set up the action to be performed on receiving the signal.

If aio_flag is 0, then no signal will be sent on completion of the operation.

When the request is first submitted, EINPROG is written to aio_errno; this value may be polled to determine when the asynchronous operation completes. If the I/O operation completes successfully, the aio_errno is set to 0, and aio_nobytes is set to the number of bytes read. If the I/O operation fails, aio_errno is set to the appropriate error code, and aio_nobytes is set to - 1. Once the call completes, the **aiocb** and any data buffers used in the call can be re-used.

The aio_prio field is currently ignored.

The **aiocbp** can be used in a subsequent acancel operation to try and cancel the request, or in a subsequent iosuspend operation, to wait for pending asynchronous operations.

DO NOT re-use the **aiocb** or the **buf** before the completion of the previous asynchronous I/O operation.

If **aiocbp** is NULL, then the read starts from the current file pointer, no signal is sent to the calling thread on completion of the operation, and the aio_nobytes and aio_errno values are not set. Note that there is no way to know that the aread call has completed (there will be no event and the values of aio_nobytes and aio_errno may not be checked). POSIX Draft 10 does not allow a value of NULL for the **aiocbp**. We recommend against using a value of NULL for the **aiocbp**.

On close of the file descriptor, **fd**, any cancelable asynchronous I/O operations associated with the file descriptor are removed. On _exit and exec all open file descriptors associated with the thread are closed, and any asynchronous I/O operation associated with the thread are removed. Non-cancelable operations are waited for, but any corresponding event notification will not occur.

Record locking for asynchronous I/O is the same as for any other I/O operation. It is important to not release the record lock until after the I/O operation it protects has completed.

An open with the O_TRUNC flag set, or creat will not cause the cancellation of any asynchronous I/O operations. (This might have interesting results if an asynchronous write is occurring at the same time as the O_TRUNC).

Asynchronous I/O operations are not inherited on fork.

NOTE

While multiple asynchronous I/O operations are being performed against a file, it is not possible to accurately determine the file pointer position. In general, all I/O performed on such a (block) file should be performed using aread, awrite, or listio. Character special files are a different matter since seeks are not allowed on them.

To use this function, include the library libc_p.a in the compilation command line.

DIAGNOSTICS

If aread succeeds, 0 is returned; if it fails, -1 is returned.

Note that the failure of the aread call where -1 is returned is an immediate failure. The aread call can also fail asynchronously, setting aio_nobytes to - 1 and aio_errno appropriately (see below). For other codes returned asynchronously, see the read manual page.

If the call fails immediately, **errno** will contain one of the following:

EINVAL	aio_event is an invalid event, or aio_whence is an invalid value, or the resulting file pointer would be invalid.

`EBADF`	**`fd`** does not refer to a valid file opened for reading.
`EAGAIN`	All the asynchronous I/O threads are currently busy; no more memory exists for creating new events; no more asynchronous I/O buffers exit for queueing the operation.
`EFAULT`	**`aiocbp`** is not a valid address. **`buf`** is not a valid address.

If the `aread` call returns successfully, but later fails, `aio_errno` is set to one of the **`errno`** values returned by the `read` call. If the `aread` call is cancelled, using `acancel`, `aio_errno` is set to `ECANCELED`. Note that if the call fails asynchronously, it will still send a signal to the calling process.

`ECANCELED`	The `aread` call was cancelled by an `acancel` call before the I/O could complete.

CONFORMANCE

`aread` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`acancel(), read(), write(), lseek(), sigaction()`
Library Functions -	`iosuspend(), awrite()`

The POSIX asynchronous I/O calls are:

System Calls -	`acancel()`
Library Functions -	`iosuspend(), aread(), awrite(), listio(), afsync().`
Example -	`aread.c`

NAME

`assert` – verify program assertion

SYNOPSIS

`#include <assert.h>`

`void assert(expression);`

DESCRIPTION

The `assert` macro puts diagnostics into a program. When it is executed, if **`expression`** is false (i.e. compares equal to zero), `assert` prints a diagnostic message to standard error and then calls abort. The diagnostic message has the following format:

`Assertion failed:` **`expression`**`, file` *xyz*`, line` *nnn*

Where:

`expression`	is the literal expression passed to `assert`.
xyz	is the name of the source file as specified by the preprocessor macro `__FILE__`.
nnn	is the source line number of the `assert` statement as specified by the preprocessor macro `__LINE__`.

The preprocessor macro `NDEBUG` is used to control whether or not assertions actually get compiled into a program: If `NDEBUG` is defined, then the assert macro is replaced by nothing. Therefore, if a source file is compiled with the `-DNDEBUG` preprocessor option, or if it contains a `#define NDEBUG` control statement before the `#include <assert.h>` control statement, then no assertions will be compiled into the resulting object file.

SEE ALSO

Library Functions - `abort()`

NAME

`atexit` – add program termination routine

SYNOPSIS

```
#include <stdlib.h>

int atexit(func)
void(*func)();
```

DESCRIPTION

`atexit` adds the function **`func`** to a list of functions to be called without arguments on normal termination of the program. Normal termination occurs by either a call to the `exit` library function or a return from main. At most 32 functions may be registered by `atexit`; the functions will be called in the reverse order of their registration.

DIAGNOSTICS

`atexit` returns 0 if the registration succeeds, otherwise it returns 1.

SEE ALSO

Library Functions - `exit()`

NAME

`atof`, `atol`,
`atoi`, `catol`,
`catoi` – convert ASCII string to a numeric value

SYNOPSIS

```
double atof(nptr)
char *nptr;

long atol(nptr)
char *nptr;

int atoi(nptr)
char *nptr;

long catol(nptr)
char *nptr;

int catoi(nptr)
char *nptr;
```

DESCRIPTION

`atof` converts a character string pointed to by **`nptr`** to a double value. The first unrecognized character ends the conversion. `atof` recognizes an optional string of white-spaced characters, then an optional sign, then a string of digits optionally containing a decimal point, then an optional `E` or `e` followed by an optionally signed integer. If the string begins with an unrecognized character, a zero is returned.

`atol` converts the character string pointed to by **`nptr`** to a long value. Leading whitespace is skipped, and a leading "`-`" causes the value to be negated. `atoi` performs similarly to `atol`, returning an **`int`** instead of a **`long`**.

`catol` and `catoi` recognize C style base specifiers. A leading "0" in the string forces `octal` conversion, and a leading "`0x`" or "`0X`" forces hexadecimal conversion.

SEE ALSO

Library Functions - `scanf()`, `strtod()`, `strtol()`

NAME

`awrite` – asynchronously write to a file files

SYNOPSIS

```
#include <sys/aio.h>

awrite(fd, buf, nbytes, aiocbp)
int fd;
char *buf;
sizet nbytes;
struct aiocb *aiocbp;
```

DESCRIPTION

`awrite` performs an asynchronous write on the file **fd**, writing **nbytes** from buffer **buf** as if by the corresponding `write` system call.

This write, however, is performed by a separate thread of execution, in parallel with computation by the calling thread. The `awrite` call will return to the calling thread before completion of the actual write operation. Thus, while the `awrite` call is asynchronously doing the write operation the calling thread continues with the next instruction. At the completion of the operation, the file pointer will be updated.

The structure **aiocb** is defined in `<sys/aio.h>`. The relevant fields are as follows (not necessarily in this order):

```
off_t          aio_offset;
int            aio_whence;
int            aio_flag;
struct event   aio_event;
int            aio_errno;
size_t         aio_nobytes;
int             aio_prio;
```

The events structure, defined in <sys/events.h> contains the following fields (not necessarily in this order):

```
evt_class_t    evt_class;
void           *evt_value;
void           (*evt_handler)() ;  /* not used by LynxOS */
evtset_t       evt_classmask ;     /* not used by LynxOS */
```

Looking ahead to POSIX Draft 10, Lynx only uses the `evt_value` and `evt_class` fields of the event structure. For the same reason, we use the words “event” and “signal” interchangeably in this document, and use the `sigaction` system call to set up the action to be take when the event is received by the calling process.

The event handler for Draft 9 is different from the handler for Draft 10. To request the Draft 9 handler, set the value of the `sigaction` `sa_flags` to `SA_D9EV`. To request the Draft 10 handler, set the value of the `sigaction` `sa_flags` to 0.

The following is a sample Draft 10 event handler:

```
void
aio_write_handler(signo, value)
int  signo;
void *value;
{
  printf("in aio_write_handler:  ");
  printf("signo = %d, value = %ld\n",signo,(long) value);
}
```

The following is a sample Draft 9 event handler:

```
void
aio_write_handler(event_value, event_class, evt_mask)
void *event_value;
evt_class_t event_class;
evtset_t evt_mask;
{
  printf("in aio_write_handler\n");
}
```

Before the data is written, an `lseek` is performed on the file **`fd`** to advance the file pointer to the desired position, using `aio_whence` and `aio_offset` as the arguments to `lseek`. However, if `O_APPEND` flag has been set by `open` or `fcntl`, then the `aio_whence` and `aio_offset` values are ignored.

If `aio_flag` contains the flag `AIO_EVENT`, then, when the asynchronous operation completes, the signal (event) `aio_event.evt_class` is sent to the calling thread, with event data `aio_event.evt_value`. If `aio_event.evt_class` is 0, no signal is sent. The calling thread can set the action to be performed on receiving the signal to an appropriate event handler (similar to a signal handler). The calling thread should use `sigaction` (system call to set up the action to be performed on receiving the signal.

If `aio_flag` is 0, then no signal will be sent on completion of the operation.

When the request is first submitted, `EINPROG` is written to `aio_errno`; this value may be polled to determine when the asynchronous operation completes. If the I/O operation completes successfully, the `aio_errno` is set to 0, and `aio_nobytes` is set to the number of bytes written. If the I/O operation fails, `aio_errno` is set to the appropriate error code, and `aio_nobytes` is set to -1. Once the call completes, the **`aiocb`** and any data buffers used in the call can be re-used.

The `aio_prio` field is currently ignored.

The **`aiocbp`** can be used in a subsequent `acancel` operation to try and cancel the request, or in a subsequent `iosuspend` operation, to wait for pending asynchronous operations. DO NOT re-use the **`aiocb`** or the **`buf`** before the completion of the previous asynchronous I/O operation.

If **`aiocbp`** is NULL, then the `write` starts from the current file pointer, no signal is sent to the calling thread on completion of the operation, and the `aio_nobytes` and `aio_errno` values are not set. Note that there is thus no way to know that the `awrite` call has completed (there will be no event and the values of `aio_nobytes` and `aio_errno` may not be checked). POSIX Draft 10 does not allow a value of NULL for the **`aiocbp`**. We recommend against using a value of NULL for the **`aiocbp`**.

Record locking for asynchronous I/O is the same as for any other I/O operation. It is important to not release the record lock until after the I/O operation it protects has completed.

On `close` of the file descriptor, **`fd`**, any cancelable asynchronous I/O operations associated with the file descriptor are removed. On `_exit` and `exec` all open file descriptors associated with the thread are closed, thus any asynchronous I/O associated with the thread are removed. Non-cancelable operations are waited for, but any corresponding event notification will not occur.

An `open` with the `O_TRUNC` flag set, or `creat` will not cause the cancellation of any asynchronous I/O operations.

Asynchronous I/O operations are not inherited on `fork`.

NOTE

While multiple asynchronous operations are being performed against a file, it is not possible to accurately determine the file pointer position. In general, all I/O performed on such a (block) file should be performed using `aread`, `awrite`, or `listio`.

Alternatively, the file may be set to append-only mode using `fcntl` to set the `O_APPEND` flag. This has the effect of pegging the seek pointer at the end of the file. Character special files are a different matter since seeks are not allowed on them.

To use this function, include the library `libc_p.a` in the compilation command line.

DIAGNOSTICS

If `awrite` succeeds, 0 is returned; if it fails, -1 is returned.

Note that this failure of the `awrite` call refers to immediate failure. The `awrite` call can also fail asynchronously, setting `aio_nobytes` to -1 and `aio_errno` appropriately (see below). For error codes returned asynchronously, see the `write` manual page.

If the call fails immediately, **`errno`** will contain one of the following:

`EAGAIN`	All the asynchronous I/O threads are currently busy; no more memory exists for creating new events or asynchronous I/O buffers for queueing the operation.

`EBADF`	**`fd`** does not refer to a valid file open for writing.
`EINVAL`	`aio_event` is an invalid event, or `aio_whence` is an invalid value, or the resulting file pointer would be invalid.
`EFAULT`	**`aiocbp`** is not a valid address. **`buf`** is not a valid address.

If the `awrite` call returns successfully, but later fails, `aio_errno` is set to one of the **`errno`** values returned by the `write` call. If the `awrite` call is cancelled, using `acancel`, `aio_errno` is set to `ECANCELED`. Note that if the call fails asynchronously, it will still send a signal to the calling process.

`ECANCELED`	The `awrite` call was cancelled by an `acancel` call before the I/O could complete.

CONFORMANCE

`awrite` is implemented according to Draft 9 of the POSIX 1003.4 (Real Time) standard.

SEE ALSO

System Calls -	`acancel(), read(), write(), lseek(), sigaction(), fcntl()`
Library Functions–	`iosuspend(), aread()`

The POSIX asynchronous I/O calls are:

System Calls -	`acancel()`
Library Functions–	`iosuspend(), aread(), awrite(), listio(), afsync().`
Example -	`awrite.c`

NAME

`basename` – get basename portion of a filename

SYNOPSIS

```
char *basename(s)
char *s;
```

DESCRIPTION

The `basename()` function returns a pointer to the basename portion of the filename **s**. The basename of a file is the name used to refer to it within its own directory. The basename is determined by deleting the last occurrence of / in the filename **s** as well as all characters that preceded it.

SEE ALSO

Utility Programs - `basename`

NAME

`bessel: j0, j1,`
`jn, y0, y1, yn` – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x)
int n;
double x;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
int n;
double x;
```

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders. The Bessel functions satisfy the recurrence relations

$$J_{n+1}(x) = \frac{2n}{x} J_n(x) - J_{n-1}(x)$$

and

$$Y_{n+1}(x) = \frac{2n}{x} Y_n(x) - Y_{n-1}(x)$$

DIAGNOSTICS

Non-positive `x` values cause `y0`, `y1`, and `yn` to return a `NaN` (Not-a-Number) and set **`errno`** to `EDOM`.

SEE ALSO

Library Functions - `exp()`

NAME

`bsearch` – binary search a sorted table

SYNOPSIS

```
char *bsearch(key, base, nel, key_size, compar)
char *key, *base;
unsigned nel;
int key_size;
int (*compar)();
```

DESCRIPTION

`bsearch` performs a binary search on a sorted table. The table should be sorted in increasing order according to the given comparison function **`compar`**. **`key`** points to a data item that is being searched for. **`base`** points to the element at the base of the table. The number of elements in the table is specified by **`nel`** and the size of each element is specified by **`key_size`**. **`compar`** points to a function which will be called to compare elements during the search. **`compar`** is invoked as follows:

```
compar(item1, item2)
itemtype *item1, *item2;
```

`compar` must return a number less than, equal to, or greater than zero, reflecting the relationship between `*item1` and `*item2`.

DIAGNOSTICS

`bsearch` returns a pointer to the element in the table that matches the **`key`**, or NULL if no match was found.

SEE ALSO

Library Functions - `qsort()`

NAME

`bstring: bcmp,`
`bcopy, bzero` – block of memory operations

SYNOPSIS

```
int bcmp(src, dst, len)
char *src, *dst;
int len;

void bcopy(src, dst, len)
char *src, *dst;
int len;

void bzero(dst, len)
char *dst;
int len;
```

DESCRIPTION

The byte string functions `bcmp`, `bcopy`, and `bzero` manipulate blocks of memory of variable length. Unlike the string functions (`strcpy`, `strcmp`, etc.), null bytes in the blocks are not important.

bcmp compares the blocks of memory starting at **src** and **dst**. Both blocks are **len** bytes long. **bcmp** returns zero if the blocks are identical, and one if they are different. `bcopy` copies **len** bytes from the block located at **src** to the block located at **dst**. `bzero` fills the block at address **dst** with **len** zeros.

The `bcopy` routine correctly handles copies of overlapping blocks of memory.

SEE ALSO

Library Functions - `strcmp(), strcpy(), memcmp(), memcpy(), memset()`

NAME

`buildgroups` – build group access list

SYNOPSIS

```
#include <conf.h>

int  buildgroups(glist, name)
int *glist;
char *name;
```

DESCRIPTION

`buildgroups` fills the array **`glist`** with the group ID numbers of all groups to which the user specified by **`name`** belongs. If **`name`** is NULL, then the current user is used. The group information is taken from the user's effective group ID and from the file `/etc/group`. Up to `NGROUPS` (defined in **`conf.h`**) entries will be placed in the array; the actual number of groups is returned. The first entry is always the user's effective group ID.

`buildgroups` is used by `login` to set up the list of groups to be given to the `setgroups` system call. Programs which must know the current actual list of groups should use the `getgroups` system call instead of `buildgroups`.

SEE ALSO

Utility Programs - `groups`

System Calls - `getegid(), getgroups(), setgroups()`

File Formats - `group`

NAME

`BUILDPID` – create a combined thread/process ID.

SYNOPSIS

```
#include <proc.h>

int combined_id;
combined_id = BUILDPID(process_id, thread_id)
int process_id;
int thread_id;

int extracted_thread_id;
int extracted_process_id;
extracted_thread_id = TIDGET(combined_id);
extracted_process_id = PIDGET(combined_id);
```

DESCRIPTION

`BUILDPID` combines a thread ID and a process ID into a combined ID that can be used as an argument to `kill`, `getpriority`, and `setpriority`.

Functions and system calls that take combined thread/process IDs built with `BUILDPID` also accept unmodified process IDs.

The thread ID part of a combined ID can be retrieved with `TIDGET`. The process ID part of a combined ID can be retrieved with `PIDGET`.

The following are always true:

```
PIDGET(BUILDPID(p,t)) == p
TIDGET(BUILDPID(p,t)) == t
```

NOTE

`BUILDPID`, `TIDGET`, and `PIDGET` may be implemented as macros.

SEE ALSO

System Calls - `getpid(), kill(), st_build()`

Library Functions - `getstid()`

NAME

`cbrt` – cube root function

SYNOPSIS

```
#include <math.h>

double cbrt(x)
double x;
```

DESCRIPTION

`cbrt()` returns the cube root of its argument. If the argument is NaN (Not a Number) or infinity, `cbrt()` returns it unchanged. `cbrt()` is accurate to within 0.666 ulps (units in the last place).

SEE ALSO

Library Functions - `sqrt()`, `pow()`

NAME

`cfgetispeed, cfgetospeed,`
`cfsetispeed, cfsetospeed` – baud rate functions

SYNOPSIS

```
#include <termios.h>

speed_t cfgetospeed(termios_p)
struct termios *termios_p;

int cfsetospeed(termios_p, speed)
struct termios *termios_p speed_t speed;

speed_t cfgetispeed(termios_p)
struct termios *termios_p;

int cfsetispeed(termios_p, speed)
struct termios *termios_p speed_t speed;
```

DESCRIPTION

These routines are provided for getting and setting the values of the input and output baud rates in the **termios** structure. The effects on the terminal device described below do not become effective until the `tcsetattr()` function is successfully called, and not all errors are detected until `tcsetattr()` is called, as well.

The input and output baud rates are represented in the **termios** structure. The values shown in the following table are defined. The name symbols in this table are defined in **<termios.h>**.

Name	Description	Name	Description
B0	Hang up	B600	600 baud
B50	50 baud	B1200	1200 baud
B75	75 baud	B1800	1800 baud
B110	110 baud	B2400	2400 baud

Name	Description	Name	Description
`B134`	134.5 baud	`B4800`	4800 baud
`B150`	150 baud	`B9600`	9600 baud
`B200`	200 baud	`B19200`	19200 baud
`B300`	300 baud	`B38400`	38400 baud

The **termios_p** argument is a pointer to a **termios** structure.

`cfgetospeed()` returns the output baud rate stored in the **termios** structure to which **termios_p** points.

`cfgetispeed()` returns the input baud rate stored in the **termios** structure to which **termios_p** points.

`cfsetospeed()` sets, to **speed**, the output baud rate stored in the **termios** structure to which **termios_p** points.

`cfsetispeed()` sets, to **speed**, the input baud are stored in the **termios** structure to which **termios_p** points.

Certain values for speeds that are set in the **termios** structure and passed to `tcsetattr()` have special meanings. These are discussed under `tcsetattr()`.

DIAGNOSTICS

`cfgetispeed()` and `cfgetospeed()` return exactly the value found in the **termios** data structure, without interpretation.

Both `cfsetispeed()` and `cfsetospeed()` return a value of zero. Both functions perform no actions if **speed** specifies an unsupported baud rate.

COPYRIGHT NOTICE

Engineers, Inc.). This information was written within the context of the ISO/IEC 9945-1:1990. The IEEE takes no responsibility or liability for and will assume no liability for any damages resulting from the reader's misinterpretation of said information resulting from the placement and context in this publication. Information is reproduced with the permission of the IEEE.

SEE ALSO

Library Functions - `tcsetattr()`

NAME

`clock` – report CPU time used

SYNOPSIS

`long clock()`

DESCRIPTION

`clock` calculates and reports the amount of CPU time (in microseconds) used since the first call to `clock`. The value returned by `clock` is the sum of the user and system times of the calling process and its terminated child processes for which it has executed `wait`, `pclose`, or `system`.

The clock resolution is 10 milliseconds.

SEE ALSO

System Calls - `times()`, `wait()`

Library Functions - `popen()`, `system()`

NAME

`creat` – create a file

SYNOPSIS

```
int creat(name, mode)
char *name;
int mode;
```

DESCRIPTION

`creat` is provided for compatibility with old software, but is functionally superseded by `open`. Any call to `creat` can be replaced with a call to `open`:

```
fd = creat(filename, mymode);
```

is identical in effect to

```
fd = open(filename, O_WRONLY | O_TRUNC | O_CREAT, mymode);
```

If a file called **`name`** exists, and its current mode allows, it is truncated to zero length and opened for writing. The current mode and owner are not changed. If on the other hand the file did not exist, it is created with mode **`mode`**. The mode for a newly-created file need not allow writing.

SEE ALSO

System Calls - `open()`, `chmod()`

NAME

crypt, setkey, encrypt – DES encryption protection

SYNOPSIS

```
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;
int edflag;
```

DESCRIPTION

`crypt` is the password encryption routine. It is based on the NBD Data Encryption Standard. `key` is normally a user's password. **salt** is a 2-character string chosen from the set [`a-zA-Z0-9./`]. The **salt** string is used to encrypt the password **key**. The value returned by `crypt` points to the encrypted password.

`setkey` and encrypt provide the actual access to the DES algorithm. The argument **key** passed to `setkey` is an array of length 64 containing the characters `0` and `1`. To obtain the 56-bit key which is used to form the new algorithm, the array is broken up into groups of eight and the eighth character in each group is dropped. This leaves a 56-character string which represents the 56-bit key used by the DES algorithm. The new algorithm is encoded in a modified bit string which is passed back in the same argument, **key**, as passed into `setkey`. Note that the key array passed into setkey must therefore be in writable memory. Code such as the following may result in segmentation violations, especially if your C compiler stores strings in the text segment of your executable programs (Gcc, the GNU C Compiler, is such a compiler):

```
char *setkey_str =
"11000011101011001100001110101100110000111010110011000011
10101100";

setkey(setkey_str);
```

The argument **`block`** to the function `encrypt` is also a 64 byte array containing the values 0 and 1. This array is modified in place to a similar array representing the bits of the original array after having been subjected to the DES algorithm using the key set by `setkey`. If **`edflag`** is 0, the argument is encrypted; otherwise, it is decrypted.

NOTE

These functions are no longer shipped with the standard LynxOS release. Domestic customers can get these functions by calling technical support.

SEE ALSO

Utility Programs - `passwd, login`

Library Functions - `getpass()`

NAME

`csem_create`,
`csem_create_val` – create a user semaphore

SYNOPSIS

```
#include <sem.h>

csem_t csem_create()

int initial_value;
csem_t csem_create_val(initial_value)
```

DESCRIPTION

`csem_create` creates a new user semaphore with initial value 1 (unlocked for the first locking thread) and returns a semaphore descriptor for same. `csem_create_val` creates a new user semaphore with initial value **`initial_value`** and returns a semaphore descriptor for same.

A counting semaphore is a synchronization mechanism that can be used by multiple threads to mediate access to shared resources. A counting semaphore is also a useful queueing mechanism. A counting semaphore, as the name implies, consists of a count (an integral value), and a (possibly empty) queue of blocked processes. If the semaphore count is positive, the semaphore is unlocked. If the semaphore count is zero, then the semaphore is locked. If the semaphore count is less than zero, then the semaphore is locked and there are threads blocked on the counting semaphore.

Waiting on a counting semaphore consists of decrementing the semaphore count. If the count becomes negative, the thread blocks and is added to the list of threads blocked on the semaphore. Otherwise, the thread continues its execution.

Signalling a counting semaphore consists of incrementing the semaphore count. If the count becomes or remains positive, then nothing else happens. If the semaphore count remains negative or becomes zero, though, one of the threads blocked on the semaphore is awakened.

A counting semaphore is different from a condition variable in that a counting semaphore has state (the count) whereas the condition variable has none. Because the condition variable is stateless, it must be used in conjunction with a mutex. A counting semaphore requires no mutex, and in fact there is no system operation coupling counting semaphores and mutexes.

A counting semaphore is different than a mutex in two respects. One, a mutex is binary: locked or unlocked. Numerous calls to `mutex_exit` do not change the state of the mutex. It simply remains locked. In fact, calling `mutex_exit` with a mutex that is not locked is an error. In contrast, numerous calls to `csem_signal` do change the state of the counting semaphore, by incrementing the count. If a counting semaphore is signalled N times, then it will need to be waited upon N times before any thread blocks as a result of waiting on the semaphore. Second, a mutex has distinct ownership and the counting semaphore does not. It is an error for any thread other than the owner of a particular mutex to try and exit that mutex.

It is acceptable for one thread to lock a counting semaphore and another thread to unlock the semaphore. Thus, semaphores are useful for general queueing of threads, whereas mutexes in and of themselves are not.

NOTE

Using a counting semaphore that has not been created with `csem_create`, or which has been deleted with `csem_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`csem_create` returns NULL if it fails to get the semaphore.

If the call fails, **`errno`** will contain one of the following:

`ENOSPC`	No more room for user semaphores.
`EINVAL`	The value provided to `csem_create_val` is less than 0.

SEE ALSO

Library Functions - `csem_delete(), csem_wait(), csem_signal(), cv_create(), mutex_create(), mutex_enter(), cv_wait(), cv_signal()`

NAME

`csem_delete` – delete a user semaphore

SYNOPSIS

```
#include <sem.h>

int csem_delete(sd)
csem_t sd;
```

DESCRIPTION

`csem_delete` frees up the system entry for the semaphore identified by **sd** and removes it from the list of active semaphores.

NOTE

Using a counting semaphore that has not been created with `csem_create`, or which has been deleted with `csem_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`csem_delete` returns 0 if the semaphore was deleted, or -1 if not. If the call fails, **errno** will contain one of the following:

EBUSY	Some thread is waiting on the semaphore (the count is negative).
EINVAL	Semaphore **sd** does not exist.

SEE ALSO

Library Functions - `csem_create()`, `csem_wait()`, `csem_signal()`, `cv_create()`, `mutex_create()`, `mutex_enter()`, `cv_wait()`, `cv_signal()`

NAME

`csem_signal` – signal a user semaphore

SYNOPSIS

```
#include <sem.h>

int csem_signal(sd)
csem_t sd;
```

DESCRIPTION

`csem_signal` increments the semaphore count for semaphore **sd**. If there are any threads waiting on the semaphore, the first one in the queue is awoken. That thread then returns from its `csem_wait` call.

A counting semaphore is a synchronization mechanism that can be used by multiple threads to mediate access to shared resources. A counting semaphore is also a useful queueing mechanism. A counting semaphore, as the name implies, consists of a count (an integral value), and a (possibly empty) queue of blocked processes. If the semaphore count is positive, the semaphore is unlocked. If the semaphore count is zero, then the semaphore is locked. If the semaphore count is less than zero, then the semaphore is locked and there are threads blocked on the counting semaphore.

Waiting on a counting semaphore consists of decrementing the semaphore count. If the count becomes negative, the thread blocks and is added to the list of threads blocked on the semaphore. Otherwise, the thread continues its execution.

Signalling a counting semaphore consists of incrementing the semaphore count. If the count becomes or remains positive, then nothing else happens. If the semaphore count remains negative or becomes zero, though, one of the threads blocked on the semaphore is awakened.

A counting semaphore is different from a condition variable in that a counting semaphore has state (the count) whereas the condition variable has none. Because the condition variable is stateless, it must be used in conjunction with a mutex. A counting semaphore requires no mutex, and in fact there is no system operation coupling counting semaphores and mutexes.

A counting semaphore is different than a mutex in two respects. One, a mutex is binary: locked or unlocked. Numerous calls to `mutex_exit` do not change the state of the mutex. It simply remains locked. In fact, calling `mutex_exit` with a mutex that is not locked is an error. In contrast, numerous calls to `csem_signal` do change the state of the counting semaphore, by incrementing the count. If a counting semaphore is signalled N times, then it will need to be waited upon N times before any thread blocks as a result of waiting on the semaphore. Second, a mutex has distinct ownership and the counting semaphore does not. It is an error for any thread other than the owner of a particular mutex to try and exit that mutex. It is acceptable for one thread to lock a counting semaphore and another thread to unlock the semaphore. Thus, semaphores are useful for general queueing of threads, whereas mutexes in and of themselves are not.

NOTE

These calls have nothing to do with the signal facilities of the operating system. Sending a signal to a process/thread and signaling a semaphore are two completely different things.

NOTE

Using a counting semaphore that has not been created with `csem_create`, or which has been deleted with `csem_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors. Worse, the application may continue to run, but simply not synchronize its operations due to the faulty counting semaphore.

DIAGNOSTICS

`csem_signal` returns 0 if successful, or -1 if not. If the call fails, **`errno`** will contain one of the following:

`EINVAL`	**`sd`** is not a valid semaphore descriptor. (This error may be returned if the application does not crash due to the invalidity of the counting semaphore).

SEE ALSO

Library Functions - `csem_create()`, `csem_wait()`, `csem_delete()`, `cv_create()`, `mutex_create()`, `mutex_enter()`, `cv_wait()`, `cv_signal()`

NAME

`csem_wait` – wait on a user semaphore

SYNOPSIS

```
#include <time.h>
#include <st.h>
#include <sem.h>

int csem_wait(sd, timeout)
csem_t sd;
struct timeval *timeout;
```

DESCRIPTION

`csem_wait` decrements the semaphore count for the semaphore given by **sd** and returns immediately if the count is non-negative. If the count is negative after the decrement performed by the call, the calling thread is placed on the wait queue of threads waiting on the semaphore and sleeps.

A counting semaphore is a synchronization mechanism that can be used by multiple threads to mediate access to shared resources. A counting semaphore is also a useful queueing mechanism. A counting semaphore, as the name implies, consists of a count (an integral value), and a (possibly empty) queue of blocked processes. If the semaphore count is positive, the semaphore is unlocked. If the semaphore count is zero, then the semaphore is locked. If the semaphore count is less than zero, then the semaphore is locked and there are threads blocked on the counting semaphore.

Waiting on a counting semaphore consists of decrementing the semaphore count. If the count becomes negative, the thread blocks and is added to the list of threads blocked on the semaphore. Otherwise, the thread continues its execution.

Signalling a counting semaphore consists of incrementing the semaphore count. If the count becomes or remains positive, then nothing else happens. If the semaphore count remains negative or becomes zero, though, one of the threads blocked on the semaphore is awakened.

A counting semaphore is different from a condition variable in that a counting semaphore has state (the count) whereas the condition variable has none. Because the condition variable is stateless, it must be used in conjunction with a mutex. A counting semaphore requires no mutex, and in fact there is no system operation coupling counting semaphores and mutexes.

A counting semaphore is different than a mutex in two respects. One, a mutex is binary: locked or unlocked. Numerous calls to `mutex_exit` do not change the state of the mutex. It simply remains locked. In fact, calling `mutex_exit` with a mutex that is not locked is an error. In contrast, numerous calls to `csem_signal` do change the state of the counting semaphore, by incrementing the count. If a counting semaphore is signalled N times, then it will need to be waited upon N times before any thread blocks as a result of waiting on the semaphore. Second, a mutex has distinct ownership and the counting semaphore does not. It is an error for any thread other than the owner of a particular mutex to try and exit that mutex. It is acceptable for one thread to lock a counting semaphore and another thread to unlock the semaphore. Thus, semaphores are useful for general queueing of threads, whereas mutexes in and of themselves are not.

While the thread is waiting for the semaphore, it can time out by using the **`timeout`** parameter. If **`timeout`** is NULL, then the thread will not time out at all. Otherwise, the thread will time out of the semaphore wait after the time indicated by **`timeval`** has passed. If **`timeout`** is zero (`timeout->tv_sec = timeout->tv_usec = 0L`), then a poll is done and the thread will not wait at all if the semaphore can not be immediately locked.

NOTE

Using a counting semaphore that has not been created with `csem_create`, or which has been deleted with `csem_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors. Worse, the application may continue to run, but simply not synchronize its operations due to the faulty counting semaphore.

DIAGNOSTICS

`csem_wait` returns 0 if the wait succeeded, or -1 if not. If the call fails, **`errno`** will contain one of the following:

`EINVAL`	**`sd`** is not a valid semaphore descriptor. (This error may be returned if the application does not crash due to the invalidity of the counting semaphore).
`EINTR`	The wait was interrupted by a signal, or timed out because the time interval indicated by **`timeout`** has passed.

SEE ALSO

Library Functions - `csem_create()`, `csem_signal()`, `csem_delete()`, `cv_create()`, `mutex_create()`, `mutex_enter()`, `cv_wait()`, `cv_signal()`

NAME

`ctermid, ctermid_r` – generate file name for terminal

SYNOPSIS

```
#include <stdio.h>

char *ctermid(s)
char *s;

char *ctermid_r(s)
char *s;
```

DESCRIPTION

`ctermid` attempts to open the file `/dev/tty` read-only and if successful, stores the string `/dev/tty` in the area pointed to by **s**, or, if **s** is NULL, in an internal static area. If **s** is not NULL, then it should point to an array of at least `ctermid` bytes. If `/dev/tty` cannot be opened, then `ctermid` stores the null string in the area pointed to by **s**, or, if **s** is NULL, in the internal static area. `ctermid` uses only one internal static area, which is overwritten be each successive call in which **s** is NULL.

If **s** is NULL, `ctermid` returns a pointer to the internal static area. Otherwise it returns **s**.

ctermid_r is a thread-reentrant version of `l_ctermid`. It is identical to `ctermid`, except that if **s** is NULL, **ctermid_r** returns NULL. **ctermid_r** uses no internal static area.

SEE ALSO

Library Functions - `isatty()`

NAME

`ctime, localtime,`
`gmtime, asctime,`
`difftime, mktime,`
`timezone, strftime,`
`tzset, tzname,`
`ctime_r, localtime_r,`
`gmtime_r, asctime_r` – interpret and convert system time values

SYNOPSIS

```
#include <time.h>

char *ctime(clock)
time_t *clock;

struct tm *localtime(clock)
time_t *clock;

struct tm *gmtime(clock)
time_t *clock;

char *asctime(tp)
struct tm *tp;

double difftime(time1, time0)
time_t time1;
time_t time0;

time_t mktime(tp)
struct tm *tp;

char *timezone(zone, dst)
int zone;
int dst;
```

POSIX SYNOPSIS

```
#include <time.h>

char *ctime(clock)
time_t *clock;
```

```
struct tm *localtime(clock)
time_t *clock;

struct tm *gmtime(clock)
time_t *clock;

char *asctime(tp)
struct tm *tp;

double difftime(time1, time0)
time_t time1;
time_t time0;

size_t strftime(buf, len, fmt, tp)
char *buf;
size_t len:
char *fmt;
struct tm *tp;

time_t mktime(tp)
struct tm *tp;

void tzset();

extern char *tzname[2];

char *ctime_r(clock, buf, len)
time_t *clock;
char *buf;
int len;

int localtime_r(result, clock)
struct tm *result;
time_t *clock;

int gmtime_r(result, clock)
struct tm *result;
time_t *clock;

int asctime_r(tm, buf, len)
struct tm *tm;
char *buf;
int len;
```

DESCRIPTION

`ctime` transforms the value pointed to by **`clock`** into a printable character string. The time value represents the number of seconds since the beginning of LynxOS time, usually midnight GMT, 1 January 1970. The result string is always twenty-six characters long, including the terminating newline and NULL, as follows:

```
Day Mon DD HH:MM:SS 19YY\n\0
```

The `Day` and `Mon` fields are always three characters long. `DD` contains the day of the month.

`localtime` returns the address of a structure containing a broken-down version of a simple second-count time value. The value is adjusted for the current system timezone and the current daylight saving time status. `gmtime` operates similarly but performs no local adjustments. The structure **`tm`** returned is defined in **`time.h`** as follows:

```
struct tm {
  int tm_sec;    /* seconds after the minute - [0,59] */
  int tm_min;    /* minutes after the hour - [0,59] */
  int tm_hour;   /* hours since midnight - [0,23] */
  int tm_mday;   /* day of the month - [1,31] */
  int tm_mon;    /* months since January - [0,11] */
  int tm_year;   /* years since 1900 */
  int tm_wday;   /* days since Sunday - [0,6] */
  int tm_yday;   /* days since January 1 - [0,365] */
  int tm_isdst;  /* Daylight Savings Time flag */
}
```

The meanings of the components and their normal ranges are expressed in the comments. The value of `tm_isdst` is positive if Daylight Savings Time is in effect, zero if Daylight Savings Time is not in effect, and negative if the information is not available. A structure returned from `gmtime` will always have `tm_isdst` set to zero.

`mktime` converts the local time in the **`tm`** structure pointed to by **`tp`** to a simple second-count time value. The original values of `tm_wday` and `tm_yday` are ignored, and the original values of the other components are not restricted to the ranges indicated above. If successful, `mktime` sets all components to represent the specified calendar time using values that are within the ranges indicated above. The values of `tm_wday` and `tm_yday` are set last, based on the new values of the other components. If the original value of `tm_isdst` is zero, `mktime` assumes that daylight savings time is not in effect; if it is greater than zero, `mktime` assumes that daylight savings time is in effect; if it is less than zero, `mktime` queries the operating system using `gettimeofday` to determine whether or not daylight savings time is in effect.

`asctime` converts the contents of a **`tm`** structure to a printable string in the same format as the result of `ctime`.

`ctime_r`, `asctime_r`, `localtime_r`, and `gmtime_r` are the thread-reentrant versions of these functions. They use the result buffers passed as parameters rather than static data areas for return values, and thus are safe to be called by multiple threads.

Using the BSD-style interface, the local time conversions are performed by examining the current system `time zone` via `gettimeofday`, and then consulting tables to determine whether Daylight Savings Time is currently in effect. Five different Daylight Savings Time styles are known, for the United States, Australia, Eastern Europe, Central (Middle) Europe, and Western Europe.

`timezone` returns a name for the timezone located **`zone`** minutes west of the prime meridian. If **`dst`** is zero, the standard name is returned, otherwise the Daylight Savings name. If no name is present in an internal table (which can be modified), the name is formed as a difference from GMT, as in `GMT-2` for a spot near the middle of the Atlantic Ocean.

Using the POSIX interface, `tzset` is used to set the local time conversion information according to the value of the environment variable `TZ`. This information is used by `ctime`, `strftime`, `mktime`, `localtime`, `ctime_r`, and `localtime_r`. All of these functions behave as if `tzset` were called automatically, so that the user generally never needs to explicitly call `tzset`. If `TZ` is absent from the environment, or if its value begins with a colon (`:`), `tzset` queries the operating system using `gettimeofday` to determine time zone and Daylight Savings Time information. The format of the `TZ` environment variable is as follows:

```
stdoffset[dst[offset][,start[/time],end[/time]]]
```

`Where:`

`std` and `dst`
: Indicates no less than three, nor more than `TZNAME_MAX`, bytes that are the designation for the standard (*std)* or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (`:`) or digits, the comma (`,`), the minus (–), the plus (+), and the null character are permitted to appear in these fields, but their meaning is unspecified.

`offset`
: Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

hh[`:`*mm*[`:`*ss*]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour is between zero and 24, and the minutes (and seconds) - if present - between zero and 59. Use of values outside these ranges causes undefined behavior. If preceded by a "–", the time zone shall be east of the Prime Meridian; otherwise it shall be west (which may be indicated by an optional preceding "+").

`rule`
: Indicates when to change to and back from summer time. The *rule* has the form:

date`/`*time*`,`*date*`/`*time*

where the first *date* describes when the change from standard to summer time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* is one of the following:

`Jn`
: The Julian day *n* (1 £ *n* £ 365). Leap days are not counted. That is, in all years—including

leap years—February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

`n` The zero-based Julian day (0 £ *n* £ 365). Leap days are counted, and it is possible to refer to February 29.

`Mm.n.d` The *d*'th day (0 £ *d* £ 6) of week *n* of month *m* of the year (1 £ *n* £ 5, 1 £ *m* £ 12, where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is Sunday.

The *time* has the same format as *offset* except that no leading sign ("−" or "+") is allowed. The default, if *time* is not given, is `02:00:00`.

`tzset` sets the external variable `tzname`:

```
extern char *tzname[2] = {"std", "dst"};
```

Where `std` is the name of the standard time zone, and `dst` is the name of the daylight savings (i.e. summer) time zone, as described previously.

`strftime` converts a time value contained in the **`tm`** structure pointed to by **`tp`** to a character string in a format specified by **`fmt`**. The character string is placed into the array pointed to by **`buf`**, which is assumed to contain room for at least **`len`** characters. If the result contains no more than **`len`** characters, `strftime` returns the number of characters produced (not including the terminating null character). Otherwise, it returns zero and the contents of the array are indeterminate. **`fmt`** is a character string that consists of conversion specifiers and text characters, reminiscent of `printf`.

Each conversion specifier consists of a **`%`** character followed by a character that determines the behavior of the conversion specifier. All ordinary characters (including the terminating null character) are copied unchanged from **`fmt`** into the result. Each conversion specifier is replaced by appropriate characters as described in the following list:

`%%`	same as `%`
`%a`	day of week, using the locale's abbreviated weekday names
`%A`	day of week, using the locale's full weekday names
`%b`	month, using the locale's abbreviated month names
`%B`	month, using the locale's full month names
`%c`	date and time as `%x %X`
`%C`	century (00-99)
`%d`	day of the month (01-31)
`%D`	date as `%m/%d/%y`
`%e`	day of the month (1-31; single digits are preceded by a blank)
`%h`	synonym for `%b`
`%H`	hour using 24-hour clock (00-23)
`%I`	hour using 12-hour clock (01-12)
`%j`	day of the year (001-366)
`%k`	hour using 24-hour clock (0-23; single digits are preceded by a blank)
`%l`	hour using 12-hour clock (1-12; single digits are preceded by a blank)
`%m`	month number (01-12)
`%M`	minute (00-59)
`%n`	same as `\n`
`%p`	locale's equivalent of the AM/PM designation associated with a 12-hour clock
`%r`	time as `%I:%M:%S %p`
`%R`	time as `%H:%M`

`%S`	second (00-59)
`%t`	same as `\t`
`%T`	time as `%H:%M:%S`
`%U`	week number of the year (00-53), where the first Sunday is the first day of week 1
`%w`	day of week (0-6), where Sunday is day 0
`%W`	week number of the year (00-53), where the first Monday is the first day of week 1
`%x`	date, using the locale's date format
`%X`	time, using the locale's time format
`%y`	year within century (00-99)
`%Y`	year, including century (for example, 1988)
%Z	time zone abbreviation

NOTE

The return values of the non-thread-reentrant versions of these functions are left in static data areas, and should therefore be copied if needed after another call.

As is evident from the multiple synopsis sections, LynxOS supports three different versions of the `ctime`-related library functions. Each version resides in a different library. The default version is BSD compatible and resides in the standard C library (`/lib/libc.a`). In order for a C program to use this version, it should be compiled (and linked) without the **–X** and **–v** options, and neither the `POSIXCC` nor the `SYSVCC` environment variable should be set.

The POSIX version resides in the POSIX library (`/lib/libc_p.a`). In order for a C program to use the POSIX version, it must be compiled either with `POSIXCC` set (to anything) in the environment or with the **–X** option:

```
cc –X [ options ... ]  file.c ...
```

When `cc` is invoked in either of these two ways, and the **-c** option is not also given, it will produce the following line to the linker:

```
ld [ options ...] /lib/init.o  file.o ... /lib/libc_p.a
```

If `cc` is invoked with the **-c** option to create an object module for later linking, then the linker line must include the POSIX library (`/lib/libc_p.a`) as shown above.

SEE ALSO

Utility Programs - `date`

System Calls - `gettimeofday()`

Library Functions - `time()`

NAME

`ctype: isalpha,`
`isupper, islower,`
`isdigit, isxdigit,`
`isalnum, isspace,`
`ispunct, isprint,`
`isgraph, iscntrl,`
`isascii, toupper,`
`tolower, _toupper,`
`_tolower, toascii` – classify or convert character values

SYNOPSIS

```
#include <ctype.h>

int isalpha(c)
int c;

int isupper(c)
int c;

. . .
```

DESCRIPTION

The following macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. `isascii` is defined on all integer values; the rest are defined only where `isascii` is true and on the single non-ASCII value `EOF` (-1).

`isalpha`	**c** is a letter.
`isupper`	**c** is an upper case letter.
`islower`	**c** is a lower case letter.
`isdigit`	**c** is a digit.
`isxdigit`	**c** is a hex digit.
`isalnum`	**c** is an alphanumeric character.

`isspace`	**c** is a space, tab, carriage return, newline, vertical tab, or formfeed.
`ispunct`	**c** is a punctuation character (neither control nor alphanumeric).
`isprint`	**c** is a printing character, code 040 (space) through 0176 (tilde).
`isgraph`	**c** is a printing character, like `isprint` except false for space.
`iscntrl`	**c** is a delete character (0177) or ordinary control character (less than 040).
`isascii`	**c** is an ASCII character, code less than 0200.

The following macros convert character-coded integer values. `toascii` is defined on all integer values. `_toupper` and `_tolower` use a lookup table and are defined only where `isascii` is true.

`toupper`	**c** is converted to upper case. Return value undefined if not `isascii(c)`.
`tolower`	**c** is converted to lower case. Return value undefined if not `isascii(c)`.
`_toupper`	**c** is converted to upper case. Return value undefined if not `isascii(c)`.
`_tolower`	**c** is converted to lower case. Return value undefined if not `isascii(c)`.
`toascii`	**c** is converted to be a valid ASCII character by stripping bits from **c**.

`toupper` and `tolower` are functions that operate on integers in the range -1 through 255, inclusive. If `toupper` is given a lower case letter, it returns the corresponding upper case letter. Otherwise, any input in the range -1 through 255 is returned unchanged. If `tolower` is given an upper case letter, it returns the corresponding lower case letter. Otherwise, any input in the range -1 through 255 is returned unchanged. Note that the macro `_toupper` works only when the value is known to be lower-case (presumably checked using `islower()`) and `_tolower` works only when the value is known to be upper-case (presumable checked using `isupper()`).

RETURN VALUE

The classification macros return zero if the character is not in the appropriate category, and non-zero (not necessarily one) if the character is.

If the argument of any of these macros or functions lies outside its domain, the result is undefined.

SEE ALSO

Library Functions - `string()`, `bstring()`, `xstring()`

NAME

`cuserid` – return ASCII string login name of the user

SYNOPSIS

```
#include <stdio.h>

char *cuserid(s)
char *s;
```

DESCRIPTION

`cuserid` stores an ASCII string representation of the login name of the owner of the current process in the area pointed to by **s**, or, if **s** is NULL, in an internal static area. If **s** is not NULL, then it should point to an array of at least `L_cuserid` bytes. If the login name cannot be found, `cuserid` returns a NULL pointer, and if **s** is not NULL, it stores the null string in the area pointed to by **s**. `cuserid` uses only one internal static area, which is overwritten be each successive call in which s is NULL.

DIAGNOSTICS

If the login name cannot be found, `cuserid` returns a NULL pointer. Otherwise, `cuserid` returns **s** (if **s** is not NULL) or a pointer to the internal static area (if **s** is NULL).

SEE ALSO

Library Functions - `getlogin()`, `getpwent()`

NAME

cv_broadcast – signal a condition to all awaiting threads

SYNOPSIS

```
#include <time.h>

#include <st.h>

#include <sem.h>

int cv_broadcast(cv)
cv_t cv;
```

DESCRIPTION

cv_broadcast is used to signal a condition's occurrence. When a cv_broadcast occurs, all threads waiting for **cv** are awakened. Those threads all then re-acquire the mutex lock they gave up in waiting for the condition variable (via an implicit mutex_enter), and then returns to the user. The caller of **cv_broadcast** need not possess the mutex lock. Only one thread at a time can posses the mutex lock. Therefore, the threads waiting on **cv** will continue their execution one by one as they acquire their mutex in turn.

A condition variable is a synchronization mechanism used by two or more threads of execution to synchronize their access to shared resources. A condition variable is so called because it is used to indicate a certain "condition", or logical predicate, regarding the shared resource. The typical use for a condition variable is for a thread to examine the logical predicate, and then, seeing that the predicate is not yet satisfied, to wait on the condition variable.

A condition variable is a stateless object. A thread performing a cv_wait will only be awakened by threads that do cv_signal or **cv_broadcast** after the waiting thread is blocked. There is no "locked" or "unlocked" state associated with a condition variable.

Because of the stateless nature of condition variables, it is unsafe in the abstract to use a condition variable without the protection of a mutex. The logical predicate must be checked while the thread has mutually exclusive access to the resource; otherwise a race condition exists, wherein the thread could check the condition and find it false, then, immediately before blocking on the condition variable, another thread could preempt and make the logical predicate true. When the first thread ran again, it would immediately block forever. This is an abstract argument, which serves to motivate why a mutex lock is required before a condition variable can be waited upon. Obviously, the mutex should be acquired before the condition variable is signalled, as well; however, it is not common practice to require this, and so we do not.

It is common practice for condition variables to be spuriously signalled, thus waking up the blocked thread when the corresponding logical predicate is not yet true. For this reason, condition variable blocking should always be done in a while loop, checking for the truth of the predicate both before and after blocking on the condition variable. The following code is the incorrect way to use a condition variable:

```
mutex_enter(m,NULL);
if (! condition_satisfied)
  cv_wait(c,m,NULL);
/*
* Now we assume that condition_satisfied is TRUE,
* and this may not be the case
*/
...        /* Modify critical shared data */
mutex_exit(m);
```

The correct way to use a condition variable is to use a `while` construct in place of an `if`:

```
mutex_enter(m,NULL);
while (! condition_satisfied)
  cv_wait(c,m,NULL);
/*
* Now we know that condition_satisfied is TRUE,
* because we checked it after waiting.
*/
...        /* Modify critical shared data */
mutex_exit(m);
```

NOTE

Using a condition variable that has not been created with `cv_create`, or which has been deleted with `cv_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors. Worse, the application may continue to run, but simply not synchronize its operations due to the faulty condition variable.

DIAGNOSTICS

`cv_broadcast` returns 0 when successful, or -1 upon error. If the call fails, **`errno`** will contain one of the following:

`EINVAL`	**`cv`** does not refer to a valid condition variable. (This error may be returned if the application does not crash due to the invalidity of the condition variable).

SEE ALSO

Library Functions - `mutex_exit()`, `mutex_create()`, `mutex_enter()`, `cv_wait()`, `cv_broadcast()`, `csem_create()`, `csem_wait()`, `csem_signal()`

NAME

`cv_create` – create a user condition variable

SYNOPSIS

```
#include <sem.h>

cv_t cv_create()
```

DESCRIPTION

`cv_create` initializes a new user condition variable and returns a condition variable descriptor for same.

A condition variable is a synchronization variable that can be used by threads to atomically release a mutex lock and wait for some condition to be satisfied (hence the name). When the condition is satisfied, presumably by some other thread, that thread will execute either `cv_signal` or `cv_broadcast`. This will cause one or all of the waiting threads, respectively, to awaken, re-acquire the mutex they released, and then return from the call. The use of the mutex ensures that only one thread accesses a shared resource at a time. The typical use for a condition variable is for a thread to examine the logical predicate, and then, seeing that the predicate is not yet satisfied, to wait on the condition variable.

A condition variable is a stateless object. A thread performing a `cv_wait` will only be awakened by threads that do `cv_signal` or `cv_broadcast` after the waiting thread is blocked. There is no “locked” or “unlocked” state associated with a condition variable.

Because of the stateless nature of condition variables, it is unsafe in the abstract to use a condition variable without the protection of a mutex. The logical predicate must be checked while the thread has mutually exclusive access to the resource; otherwise a race condition exists, wherein the thread could check the condition and find it false, then, immediately before blocking on the condition variable, another thread could preempt and make the logical predicate true. When the first thread ran again, it would immediately block forever. This is an abstract argument, which serves to motivate why a mutex lock is required before a condition variable can be waited upon. Obviously, the mutex should be acquired before the condition variable is signalled, as well; however, it is not common practice to require this, and so we do not.

It is common practice for condition variables to be spuriously signalled, thus waking up the blocked thread when the corresponding logical predicate is not yet true. For this reason, condition variable blocking should always be done in a while loop, checking for the truth of the predicate both before and after blocking on the condition variable. The following code is the incorrect way to use a condition variable:

```
mutex_enter(m,NULL);
if (! condition_satisfied)
  cv_wait(c,m,NULL);
/*
* Now we assume that condition_satisfied is TRUE,
* and this may not be the case
*/
...        /* Modify critical shared data */
mutex_exit(m);
```

The correct way to use a condition variable is to use a `while` construct in place of an `if`:

```
mutex_enter(m,NULL);
while (! condition_satisfied)
  cv_wait(c,m,NULL);
/*
* Now we know that condition_satisfied is TRUE,
* because we checked it after waiting.
*/
...        /* Modify critical shared data */
mutex_exit(m);
```

NOTE

Using a condition variable that has not been created with `cv_create`, or which has been deleted with `cv_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`cv_create` returns NULL if it fails to get the condition variable. If the call fails, **errno** will contain one of the following:

ENOSPC	No more room for user condition variables.

SEE ALSO

Library Functions - `cv_delete()`, `cv_wait()`, `cv_signal()`, `mutex_create()`, `mutex_enter()`, `mutex_exit()`, `csem_create()`, `csem_wait()`, `csem_signal()`

NAME

`cv_delete` – delete a user condition variable

SYNOPSIS

```
#include <sem.h>

int cv_delete(cd)
cv_t cd;
```

DESCRIPTION

`cv_delete` frees up the system entry for the condition variable identified by **cd** and removes it from the list of active condition variables.

NOTE

Using a condition variable that has not been created with `cv_create`, or which has been deleted with `cv_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`cv_delete` returns 0 if the condition variable was deleted, or -1 if not. If the call fails, **errno** will contain one of the following:

EBUSY	Some thread is waiting on the condition variable.
EINVAL	Condition variable **cd** does not exist.

SEE ALSO

Library Functions - `cv_create()`, `cv_wait()`, `cv_signal()`, `cv_broadcast()`, `mutex_create()`, `mutex_enter()`, `mutex_exit()`, `csem_create()`, `csem_wait()`, `csem_signal()`

NAME

cv_signal – signal a condition thread

SYNOPSIS

```
#include <time.h>

#include <st.h>

#include <sem.h>

int
cv_signal(cv)
cv_t cv;
```

DESCRIPTION

cv_signal is used to signal a condition's occurrence. When a cv_signal occurs, the thread waiting for **cv** to be signalled that is of equal or greater priority than all other waiters and has been waiting the longest, is awakened. That thread then re-acquires the mutex lock it gave up in waiting for the mutex (via an implicit mutex_enter), and then returns to the user. The caller of **cv_signal** need not possess the mutex lock.

A condition variable is a synchronization mechanism used by two or more threads of execution to synchronize their access to shared resources. A condition variable is so called because it is used to indicate a certain "condition", or logical predicate, regarding the shared resource. The typical use for a condition variable is for a thread to examine the logical predicate, and then, seeing that the predicate is not yet satisfied, to wait on the condition variable.

A condition variable is a stateless object. A thread performing a cv_wait will only be awakened by threads that do **cv_signal** or cv_broadcast after the waiting thread is blocked. There is no "locked" or "unlocked" state associated with a condition variable.

Because of the stateless nature of condition variables, it is unsafe in the abstract to use a condition variable without the protection of a mutex. The logical predicate must be checked while the thread has mutually exclusive access to the resource; otherwise a race condition exists, wherein the thread could check the condition and find it false, then, immediately before blocking on the condition variable, another thread could preempt and make the logical predicate true. When the first thread ran again, it would immediately block forever. This is an abstract argument, which serves to motivate why a mutex lock is required before a condition variable can be waited upon. Obviously, the mutex should be acquired before the condition variable is signalled, as well; however, it is not common practice to require this, and so we do not.

It is common practice for condition variables to be spuriously signalled, thus waking up the blocked thread when the corresponding logical predicate is not yet true. For this reason, condition variable blocking should always be done in a while loop, checking for the truth of the predicate both before and after blocking on the condition variable. The following code is the incorrect way to use a condition variable:

```
mutex_enter(m,NULL);
if (! condition_satisfied)
  cv_wait(c,m,NULL);
/*
* Now we assume that condition_satisfied is TRUE,
* and this may not be the case
*/
...        /* Modify critical shared data */
mutex_exit(m);
```

The correct way to use a condition variable is to use a `while` construct in place of an `if`:

```
mutex_enter(m,NULL);
while (! condition_satisfied)
  cv_wait(c,m,NULL);
/*
* Now we know that condition_satisfied is TRUE,
* because we checked it after waiting.
*/
...        /* Modify critical shared data */
mutex_exit(m);
```

NOTE

Using a condition variable that has not been created with `cv_create`, or which has been deleted with `cv_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors. Worse, the application may continue to run, but simply not synchronize its operations due to the faulty condition variable.

DIAGNOSTICS

`cv_signal` returns 0 when successful, or -1 upon error. If the call fails, **`errno`** will contain one of the following:

`EINVAL`	**`cv`** does not refer to a valid condition variable. (This error may be returned if the application does not crash due to the invalidity of the condition variable).

SEE ALSO

Library Functions - `mutex_exit()`, `mutex_create()`, `mutex_enter()`, `cv_wait()`, `cv_broadcast()`, `csem_create()`, `csem_wait()`, `csem_signal()`

NAME

`cv_wait` – wait for a condition to be signaled

SYNOPSIS

```
#include <time.h>
#include <st.h>
#include <sem.h>

int cv_wait(cv, mutex, timeout)
cv_t cv;
mutex_t mutex;
struct timeval *timeout;
```

DESCRIPTION

`cv_wait` is used by the owner of mutex **`mutex`** to wait for a condition to be signalled by another thread. The mutex ownership is released and the thread blocks awaiting a `cv_broadcast` or `cv_signal`.

The **`timeout`** argument specifies a maximum amount of time to wait for the condition variable to be signalled. If the argument is NULL, the thread blocks indefinitely. If the argument refers to a time of duration 0, the thread will not block at all.

Once the thread is awakened by the signalling of the condition or by a **`timeout`**, the thread performs an implicit `mutex_enter` to regain the mutex lock it gave up when it performed the `cv_wait`.

A condition variable is a synchronization mechanism used by two or more threads of execution to synchronize their access to shared resources. A condition variable is so called because it is used to indicate a certain "condition", or logical predicate, regarding the shared resource. The typical use for a condition variable is for a thread to examine the logical predicate, and then, seeing that the predicate is not yet satisfied, to wait on the condition variable.

A condition variable is a stateless object. A thread performing a `cv_wait` will only be awakened by threads that do `cv_signal` or `cv_broadcast` after the waiting thread is blocked. There is no "locked" or "unlocked" state associated with a condition variable.

Because of the stateless nature of condition variables, it is unsafe in the abstract to use a condition variable without the protection of a mutex. The logical predicate must be checked while the thread has mutually exclusive access to the resource; otherwise a race condition exists, wherein the thread could check the condition and find it false, then, immediately before blocking on the condition variable, another thread could preempt and make the logical predicate true. When the first thread ran again, it would immediately block forever. This is an abstract argument, which serves to motivate why a mutex lock is required before a condition variable can be waited upon. Obviously, the mutex should be acquired before the condition variable is signalled, as well; however, it is not common practice to require this, and so we do not.

It is common practice for condition variables to be spuriously signalled, thus waking up the blocked thread when the corresponding logical predicate is not yet true. For this reason, condition variable blocking should always be done in a while loop, checking for the truth of the predicate both before and after blocking on the condition variable. The following code is the incorrect way to use a condition variable:

```
mutex_enter(m,NULL);
if (! condition_satisfied)
  cv_wait(c,m,NULL);
/*
* Now we assume that condition_satisfied is TRUE,
* and this may not be the case
*/
...        /* Modify critical shared data */
mutex_exit(m);
```

The correct way to use a condition variable is to use a `while` construct in place of an `if`:

```
mutex_enter(m,NULL);
while (! condition_satisfied)
  cv_wait(c,m,NULL);
/*
* Now we know that condition_satisfied is TRUE,
* because we checked it after waiting.
*/
...        /* Modify critical shared data */
mutex_exit(m);
```

BEHAVIOR WITH RESPECT TO THREAD CANCELLATION

A call to `cv_wait` is a thread cancellation point. That is, if a thread has a set general cancelability and not an asynchronous cancelability, and if a cancellation request is issued against this thread before or during the duration of the call to `cv_wait`, the thread will be canceled.

NOTE

Using a condition variable that has not been created with `cv_create`, or which has been deleted with `cv_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`cv_wait` returns 0 when successful, or -1 upon error. If the call fails, **errno** will contain one of the following:

EINVAL	**cv** does not refer to a valid condition variable.
EINVAL	**mutex** does not refer to a valid mutex.
EPERM	**mutex** is valid but not owned by the caller.
EINTR	The **cv** wait was interrupted by a signal delivered to the thread.
EINTR	The **cv** wait timed out without receiving a `cv_signal`. In any case, the mutex **mutex** is reacquired.

NOTE

In any case, no matter what the success or failure return is of `cv_wait`, the mutex **mutex** is always re-acquired.

SEE ALSO

Library Functions - `mutex_exit()`, `mutex_create()`, `mutex_enter()`, `cv_signal()`, `cv_broadcast()`, `st_cancel()`, `st_setcancel()`, `st_testcancel()`, `csem_create()`, `csem_wait()`, `csem_signal()`

NAME

`difftime` – computes the difference between two times.

SYNOPSIS

```
#include <time.h>

double difftime(t1, t0)
time_t t1;
time_t t0;
```

DESCRIPTION

The `difftime()` function returns the number of seconds between **t0** and **t1** expressed as a double.

NAME

`directory: opendir, readdir, telldir, seekdir, rewinddir, closedir` – directory file operations

SYNOPSIS

```
#include <dir.h>

DIR *opendir(filename)
char *filename;

struct direct *readdir(dirp)
DIR *dirp;

long telldir(dirp)
DIR *dirp;

seekdir(dirp, loc)
DIR *dirp;
long loc;

rewinddir(dirp)
DIR *dirp;

closedir(dirp)
DIR *dirp;
```

DESCRIPTION

The directory routines provide a means of reading records from LynxOS directory files. `opendir` opens the directory file named by **`filename`**. The return value points to a structure which will later be used to identify the directory "stream." `opendir` returns NULL if the directory cannot be accessed or insufficient memory is available for buffer space.

Once a directory has been opened, `readdir` will return the next entry in the directory in a direct structure. `readdir` returns NULL at end-of-file or if it detects an earlier invalid `seekdir`.

`telldir` returns the current location in the given open directory. `seekdir` sets the position for the next `readdir` on the open directory. Valid locations for `seekdir` can be obtained from calls to `telldir`. Because of compaction performed automatically by the operating system, a location returned by `telldir` is only valid for the time the directory is open. Once the directory is closed, values obtained with `telldir` should be considered invalid.

`rewinddir` sets the position for the next `readdir` to the first entry in the directory.

`closedir` closes the directory file and frees space allocated for buffers.

SEE ALSO

Utility Programs -	`ls`
System Calls -	`open(), close(), read(), lseek()`
File Formats -	`directory`

NAME

`directory: opendir, readdir, rewinddir, closedir, opendir_r, readdir_r, closedir_r` – directory file operations (POSIX.4a)

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(filename)
char *filename;

struct dirent *readdir(dirp)
DIR *dirp;

void rewinddir(dirp)
DIR *dirp;

closedir(dirp)
DIR *dirp;

DIR *opendir_r(filename, dirp)
char *filename;
DIR *dirp;

struct dirent *readdir_r(dirp, result)
DIR *dirp;
struct dirent *result;

closedir_r(dirp)
DIR *dirp;
```

DESCRIPTION

The directory routines provide a means of reading records from LynxOS directory files. `opendir` opens the directory file named by **`filename`**. The return value points to a structure which will later be used to identify the directory "stream." `opendir` returns NULL if the directory cannot be accessed or insufficient memory is available for buffer space.

Once a directory has been opened, `readdir` will return the next entry in the directory in a direct structure. `readdir` returns NULL at end-of-file or if it detects an earlier invalid `seekdir`.

`rewinddir` sets the position for the next `readdir` to the first entry in the directory.

`closedir` closes the directory file and frees space allocated for buffers.

`opendir_r` is a thread-reentrant version of `opendir` that does not perform any internal allocation. Correspondingly, `closedir_r` is used to close a directory opened with `opendir_r`. `closedir_r` will not free the memory associated with **`dirp`**. `readdir_r` is a thread-reentrant version of `readdir` and returns its result in **`result`** rather than a system-allocated buffer.

SEE ALSO

Utility Programs -	`ls`
System Calls -	`open(), close(), read(), lseek()`
File Formats -	`directory`

NAME

`div, ldiv` – compute the quotient and remainder

SYNOPSIS

```
#include <stdlib.h>

div_t div(numer, denom)
int numer;
int denom;

ldiv_t ldiv(numer, denom)
long numer;
long denom;
```

DESCRIPTION

The `div` function computes the quotient and remainder of the division of the numerator **numer** by the denominator **denom**. It returns a structure of type `div_t`, comprising both the quotient and remainder:

```
int quot;   /* quotient */
int rem;    /* remainder */
```

If the division is inexact, **quot** will be the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise **quot * denom + rem** will equal **numer**. This definition implies that **rem** always has the same sign as **numer**.

The `ldiv` function is similar to the `div` function, except that the arguments and the members of the returned structure (which has type `ldiv_t`) all have type long.

NOTE

The `div` and `ldiv` functions exist because the built-in operators for division (`/`) and remainder (`%`) are not well defined according to the ANSI C Standard. Specifically, if either operand given to the / operator is negative, it is implementation-defined whether the quotient is the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient. Similarly, the sign of the result of the `%`

operator is implementation-defined when either operand is negative. The only thing guaranteed by ANSI is the relation `(a/b)*b + a%b = a`, assuming the quotient `a/b` is representable.

NAME

drand48, erand48,
lrand48, nrand48,
mrand48, jrand48,
srand48, seed48,
lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

```
double drand48 ()

double erand48(xsubi)
unsigned short xsubi[3];

long lrand48()

long nrand48(xsubi)
unsigned short xsubi[3];

long mrand48()

long jrand48(xsubi)
unsigned short xsubi[3];

void srand48(seedval)
long seedval;

unsigned short *seed48(seed16v)
unsigned short seed16v[3];

void lcong48(param)
unsigned short param[7];
```

DESCRIPTION

These functions generate pseudo-random numbers using the linear congruence algorithm and 48-bit integer arithmetic. All the functions generate a sequence of 48-bit integer values, Xi according to the linear congruence formula

$$X_{n+1} = (aX_n + c) \bmod m \quad n >= 0.$$

Here m=248, so 48-bit integer arithmetic is performed. The default values for a and c are as follows:

```
a=0x5deece66d
c=0xb
```

Some traits associated with these functions are:

- drand48 and erand48 functions generate non-negative double-precision floating point values uniformly distributed over the interval [0.0, 1.0].
- lrand48 and nrand48 functions generate non-negative long integers uniformly distributed over the interval [0, 2^{31}].
- mrand48 and jrand48 functions generate signed long integers uniformly distributed over the interval [-2^{31}, 2^{31}].
- srand48, seed48, and lcong48 are the initialization functions for setting the value of Xi in the formula. One of these initialization functions should be called before either drand48, lrand48, or mrand48 is invoked.

All of the functions drand48, erand48, lrand48, nrand48, mrand48, and jrand48 generate the next 48-bit Xi in the sequence. Then, depending on the type of data to be returned by the function, the appropriate number of bits are used from the higher-order bits of Xi to create the return value.

The functions drand48, lrand48, and mrand48 must be initialized prior to being invoked, because they store the last 48-bit Xi generated in an internal buffer. The functions erand48, nrand48, and jrand48 do not have to be initialized, because the calling program provides the Xi value in the **xsubi** array.

The initializer function srand48 sets the high-order 32 bits of Xi to **seedval**. The low-order 16 bits of Xi are set to the arbitrary value 0x330e.

The initializer function seed48 sets the value of Xi to the 48-bit value specified in **seed16v**. In addition, the previous value of Xi is copied into a 48-bit internal buffer, a pointer to which is returned. This pointer can be used to restart a sequence at a specific point by reinitializing seed48 with a value retrieved using the pointer.

The initialization function `lcong48` allows the user to specify the initial `Xi`, the multiplier value `a`, and the addend value `c`. `param[0]` through `c.param[2]` specify `Xi`, `param[3]` through `param[5]` specify `a`, and `param[6]` specifies `c`. After `lcong48` has been called, a subsequent call to either `srand48` or `seed48` will restore the default values for `a` and `c`, specified above.

SEE ALSO

Library Functions - `rand(), random()`

NAME

`econvert, fconvert,`
`gconvert` – convert floating-point number to a string

SYNOPSIS

```
char *econvert(value, ndigit, decpt, sign, buf)
double value;
int ndigit;
int *decpt;
int *sign;
char *buf;

char *fconvert(value, ndigit, decpt, sign, buf)
double value;
int ndigit;
int *decpt;
int *sign;
char *buf;

char *gconvert(value, ndigit, trailing, buf)
double value;
int ndigit;
int trailing;
char *buf;
```

DESCRIPTION

The `econvert` function converts the double value to a NULL- terminated string of **ndigit** ASCII digits in **buf** and returns a pointer to **buf**. The least significant digit is rounded and the most-significant digit is non-zero, unless the value is zero. The space pointed to by **buf** should contain at least **ndigit+1** characters. The position of the decimal point is stored in the integer pointed to by **decpt**. This position is relative to the beginning of the string, so that a negative value means to the left of the returned digits. If value is negative, then the integer pointed to by sign will be non-zero; otherwise it will be zero.

The `fconvert` function is similar to `econvert`, except that the appropriate digit has been rounded for the fixed-decimal format of `sprintf(%`*w*`.`*n*`f)` where *n*=**ndigit** digits to the right of the decimal point. If **ndigit** is negative, then the rounding occurs to the left of the decimal point. The return value is a pointer to **`buf`**, which should contain at least `310+max(0,`**`ndigit`**`)` characters to accommodate any double-precision value.

The `gconvert` function converts the double **`value`** to a NULL- terminated ASCII string in **`buf`** and returns a pointer to **`buf`**. If possible, `gconvert` produces **`ndigit`** significant digits in fixed-decimal format, like `sprintf(%`*w.n*`f)`, and otherwise in floating-decimal format, like `sprintf(%`*w.n*`e)`. The returned string contains any sign, decimal, or exponent, as appropriate. If **`trailing`** `= 0`, trailing zeroes and a trailing decimal point are suppressed, as in `sprintf(%g)`. If **`trailing`** `!= 0`, trailing zeroes and a trailing decimal point are retained, as in `sprintf(%#g)`.

Each of these functions handles IEEE Infinities and Not a Number by returning `"Infinity"` or `"Inf"` for Infinity, and `"NaN"` for Not a Number. The longer form of Infinity is used when **`ndigit`** `>= 8`.

These functions are compatible with the SunOS functions of the same names. The `econvert` and `fconvert` functions are re-entrant versions of `ecvt` and `fcvt`, respectively, because they create the string in the space pointed to by **`buf`**, rather than in a static data area. The `gconvert` function is a more flexible version of `gcvt`, because it allows trailing zeroes and decimal point to be retained if so desired.

SEE ALSO

Library Functions - `ecvt()`, `printf()`

NAME

`ecvt`, `fcvt`, `gcvt` – convert floating-point number to a string

SYNOPSIS

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit;
int *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit;
int *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

DESCRIPTION

`ecvt` converts the double **value** to a string of **ndigit** digits and returns a pointer to the null-terminated result. The low-order digit is rounded and the high-order digit is non-zero, unless the value is zero. The position of the decimal point is stored in the integer pointed to by **decpt**. This position is relative to the beginning of the string, so that a negative value means to the left of the returned digits. If **value** is negative, then the integer pointed to by **sign** will be non-zero, otherwise it will be zero.

`fcvt` is similar to `ecvt` except that the appropriate digit has been rounded for `printf %f` output of the number of digits specified by **ndigit**.

`gcvt` converts **value** to a string, places the null-terminated result in the array pointed to by **buf**, and returns **buf**. If possible, `gcvt` produces **ndigit** significant digits in `printf %f` format, otherwise `%e` format. A minus sign, if the result is negative, or a decimal point will be included as part of the returned string. Trailing zeroes are suppressed.

NOTE

The values returned by `ecvt` and `fcvt` point to a single static data array whose content is overwritten by each call.

SEE ALSO

Library Functions - `printf()`

NAME

`_end, _etext, _edata` – last locations in program

SYNOPSIS

`extern _etext;`

`extern _edata;`

`extern _end;`

DESCRIPTION

The symbols `_etext`, `_edata`, and `_end` are reserved for special use by the compilation and linking system. Only the addresses of these name are meaningful.

`_etext`	The address of `_etext` is the first address above the program text.
`_edata`	The address of `_edata` is the first address above the initialized data region.
`_end`	The address of `_end` is the first address above the uninitialized data region.

SEE ALSO

Utility Programs -	`gcc`
System Calls -	`brk`
Library Functions -	`malloc`

NOTE

When execution begins, the program break (the first location beyond the data) coincides with `_end`, but the program break may be reset by the routines `brk()`, `malloc()`, the standard input/output library, the profile (**–p**) option of `gcc`, etc. See `brk(2)` for more information.

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf(x)
double x;

double erfc(x)
double x;
```

DESCRIPTION

erf computes the error function of **x**. The error function is computed using the formula:

$$erf(x) = \left(\frac{2}{\sqrt{\pi}}\right)\int_0^x e^{-t^2}dt$$

erfc computes the complementary error function. This is computed using the formula:

$$erfc(x) = 1.0 - erf(x)$$

The complementary error function is provided because of the extreme loss of relative accuracy if erf is called for large **x** and the result is subtracted from 1.0.

SEE ALSO

Library Functions - exp(), sqrt()

NAME

`evtemptyset,evtfillset,`
`evtaddset,evtdelset,`
`evtismember` – manipulate event class sets

SYNOPSIS

```
#include &lt;events.h>

int evtemptyset(set)
evtset_t *set;

int evtfillset(set)
evtset_t *set;

int evtaddset(set,class)
evtset_t *set;
evt_class_t class;

int evtdelset(set,class)
evtset_t *set;
evt_class_t class;

int evtismember(set,class)
evtset_t *set;
evt_class_t class;
```

DESCRIPTION

These functions manipulate sets of event classes. They operate on data objects addressable by the application, not on any set of event classes known to the system.

The `evtemptyset()` function initializes the event class set pointed to by the argument **set** such that all event classes `EVTCLASS_MIN` through `EVTCLASS_MAX` are excluded.

The `evtfillset()` function initializes the event class set pointed to by the argument **set**, such that all event class `EVTCLASS_MIN` through `EVTCLASS_MAX` are included.

The `evtaddset()` and `evtdelset()` functions respectively add and delete event class specified by the value of the argument **`class`** from the event class set pointed to by the argument **`set`**.

The `evtismember()` function tests whether the event class specified by the value of the argument **`class`** is a member of the set pointed to by the argument **`set`**.

Upon successful completion, the `evtismember()` function returns a value of one if the specified event class is a member of the specified set, or a value of zero if it is not. Upon successful completion, the other functions return a value of zero. For all the above functions, if an error is detected, a value of -1 is returned and **`errno`** is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `evtaddset()`, `evtdelset()`, and `evtismember()` functions will return -1 and set **`errno`** to the corresponding value:

`EINVAL`	The value of class is greater than `EVTCLASS_MAX` or less than `EVTCLASS_MIN`.

SEE ALSO

Library Functions - `evtprocmask(), evtsuspend()`

NAME

`evtpoll` – poll for event notification

SYNOPSIS

```
#include <events.h>
#include <timers.h>

int evtpoll(evtmask,timeout,value,class)
evtset_t evtmask;
struct timespec *timeout;
void **value;
evt_class_t *class;
```

DESCRIPTION

The `evtpoll()` function polls for events which are currently or may subsequently be queued to the calling process. The argument **evtmask** specifies the event classes to be considered in the poll.

When the process issues the `evtpoll()` request, the **evtmask** is compared against the set of queued events which have previously arrived. If a there is match, then the event is dequeued and the function immediately returns with value containing the value associated with the event and **class** containing the class of the event.

If no events are eligible for notification, `evtpoll()` will wait for a period of time determined by the value of the argument **timeout**:

`(struct timespec *) NULL`
: The `evtpoll()` function will wait indefinitely for an event notification in one of the classes specified by **evtmask** to occur.

any valid pointer
: The `evtpoll()` function will wait for an event notification in one of the classes specified by **evtmask** for the time interval specified in the **timespec** structure pointed to by **timeout**. If the **timeout** is zero-valued

then `evtpoll()` will return immediately with an error if there are no events pending in any of the classes specified by **`evtmask`**.

If an event which is not being monitored by `evtpoll()` occurs, then its event handler is invoked asynchronously and `evtpoll()` will continue the wait.

Upon successful completion `evtpoll()` will return a zero. Otherwise a value of -1 will be returned and **`errno`** is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `evtpoll()` function returns -1 and set **`errno`** to the corresponding value:

`EAGAIN`	No event in a class specified by **`evtmask`** occurred within the specified **`timeout`** period.
`EINTR`	A signal is caught by the calling process and control is returned from the signal-catching function.
`EINVAL`	The **`timeout`** argument specified a nanosecond value less than zero or greater than or equal to 100 million.

CONFORMANCE

`evtpoll` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

`evtpoll` is implemented according to Draft 10 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Library Functions - `evtsuspend()`

Example - `evtpoll.c`

NAME

`evtprocmask` – examine and change blocked events

SYNOPSIS

```
#include <events.h>

int evtprocmask(how,set,oset)
int how;
evtset_t *set;
evtset_t *oset;
```

DESCRIPTION

The `evtprocmask()` function is used to examine and/or change the calling process' event class mask. If the value of the argument **set** is not NULL, it points to a set of event classes to be used to change the process current event class mask.

The value of the argument **how** indicates the manner in which the set is changed, and can take one of the following values:

`EVT_BLOCK`	The resulting set shall be the union of the current set and the event set pointed to by the argument **set**.
`EVT_UNBLOCK`	The resulting set shall be the intersection of the current set and the complement of the event set pointed to by the argument **set**.
`EVT_SETMASK`	The resulting set shall be the event set pointed to by the argument **set**.

If the argument **oset** is not NULL, the previous mask is stored in the space pointed to by **oset**. If the value of the argument **set** is NULL, the value of the argument **how** is not significant and the process' event class mask is unchanged.

If the `evtprocmask()` function is called from within an event notification function to change the process' event class mask, the mask installed by this call remains in effect only until the event notification function returns, or executes the `evtlongjmp()` call. When the event notification function returns or calls `evtlongjmp()` the event class mask saved on entry to the event notification function or by the corresponding `evtsetjmp()` call is restored.

Upon successful completion a value of zero is returned. Otherwise, a value of -1 is returned and **`errno`** is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `evtprocmask()` function will return -1 and set **`errno`** to the corresponding value:

`EINVAL` The value of the **`how`** argument is not equal to one of the defined values.

SEE ALSO

Library Functions - `evtsuspend()`

NAME

`evtraise` – generate an application-defined event

SYNOPSIS

```
#include <events.h>

int evtraise(eventp)
struct event *eventp;
```

DESCRIPTION

The `evtraise()` function causes the specified **eventp** to be raised for the calling process. If the event class specified in **eventp** is currently blocked, the call to `evtraise()` will return immediately and the event will be left pending. If the specified event class is not currently blocked, the event notification function for that event will be executed before returning from the event.

Upon successful completion, the `evtraise()` function returns a value of 0 indicating the specified event has been raised and, possibly, serviced. Otherwise a value of -1 is returned and **errno** is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `evtraise()` function shall return -1 and set **errno** to the corresponding value.

`EINVAL`	The value of class is greater than `EVTCLASS_MAX` or less than `EVTCLASS_MIN`

CONFORMANCE

`evtraise` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

`evtraise` is implemented according to Draft 10 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Library Functions - `evtprocmask()`

Example - `evtraise.c`

NAME

`evtsetjmp`, `evtlongjmp` – non-local goto with POSIX.4 Draft 9 event mask save/restore

SYNOPSIS

```
#include <setjmp.h>

int evtsetjmp(env)
evtjmp_buf_t env;
int evtlongjmp(env, returnval)
evtjmp_buf_t env;
int returnval;
```

DESCRIPTION

`evtsetjmp` and `evtlongjmp` do what `setjmp` and `longjmp` do. That is, `evtsetjmp` is used to record a snapshot of the machine environment at a particular point in time, and `evtlongjmp` can then be used, later, to return to that exact machine state. `evtsetjmp` and `evtlongjmp` are useful for quickly unwinding long sequences of computations which have gone horribly awry.

`evtsetjmp` and `evtlongjmp` do more than `setjmp` and `longjmp`. In particular, `evtsetjmp` records the POSIX.4 Draft 9 event mask of the process as part of the machine snapshot. Then, when `evtlongjmp` is called, the event mask recorded in the buffer is re-installed as part of the process of rewinding back to where the process was at the time `evtsetjmp` was called.

When `evtsetjmp` is directly called, it returns `0`. When `evtlongjmp` is called, it never returns; but rather, execution proceeds as if the corresponding `evtsetjmp` call had just returned. In this case, the `evtsetjmp` call seems to return the value that was passed to `evtlongjmp` as the **`returnval`** parameter.

If **`returnval`** is `0`, then `evtsetjmp` will seem to return `1`. It is impossible to fake out `evtsetjmp` so that a return via `evtlongjmp` returns `0`.

NOTE

If you call `evtlongjmp` with an uninitialized environment snapshot, or one for which the stack no longer exists (i.e., not a call frame that still exists on the stack), then unpredictable (but predictably bad) results will ensue.

`evtsetjmp` and `evtlongjmp` are defined by POSIX.4, Draft 9.

SEE ALSO

System Calls -	`sigprocmask()`
Library Functions -	`setjmp(), longjmp(), sigsetjmp(), siglongjmp()`

NAME

`evtsuspend` – wait for an event

SYNOPSIS

```
#include <events.h>
#include <timers.h>

int evtsuspend(evtmask,timeout)
evtset_t *evtmask;
timespec *timeout;
```

DESCRIPTION

The `evtsuspend()` function replaces the process' event class mask with the set of event classes pointed to by the argument **`evtmask`**. It then suspends the process until an event notification function for one of the classes unblocked by **`evtmask`** is executed, or until the specified **`timeout`** elapses.

The length of the time for which `evtsuspend()` will wait for event notification in one of the specified classes is governed by the value of the argument **`timeout`**.

(`struct timespec *`) NULL
: The `evtsuspend()` function will wait indefinitely for an event notification in one of the classes specified by **`evtmask`** to occur.

any valid pointer
: The `evtsuspend()` function will wait for an event notification in one of the classes specified by **`evtmask`** for the time interval specified in the **`timespec`** structure pointed to by **`timeout`**.

Upon successful completion, `evtsuspend()` will return a zero. Otherwise a value of -1 will be returned and **`errno`** is set to indicate the error.

NOTE

This `evtsuspend()` function will use the `SIGALRM` signal and the real time timer if a time interval is specified. The signal handler for `SIGALRM` is saved on entry and restored on exit to `evtsuspend()`. The real-time timer is suspended for the duration of `evtsuspend()`.

DIAGNOSTICS

If any of the following conditions occur, the `evtsuspend()` function returns -1 and set **`errno`** to the corresponding value:

`EAGAIN`	No event notification in a class specified by **`evtmask`** occurred within the specified **`timeout`** period.
`EINTR`	A signal is caught by the calling process prior to the execution of any event handler unblocked by **`evtmask`** and control is returned from the signal- catching function.
`EINVAL`	The **`timeout`** argument specified a nanosecond value less than zero or greater than or equal to 100 million.

CONFORMANCE

`evtsuspend` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

`evtsuspend` is implemented according to Draft 10 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Library Functions - `pause(), evtprocmask()`

Example - `evtsuspend.c`

NAME

execl, execv,
execle, execlp,
execvp – simplified loader interface

SYNOPSIS

```
execl(name, arg0, arg1, ..., argn, (char *) 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, args)
char *name, **args;

execle(name, arg0, arg1, ..., argn, (char *) 0, envp)
char *name, *arg0, *arg1, ..., *argn, **envp;

execlp(name, arg0, arg1, ..., argn, (char *) 0)
char *name, *arg0, *arg1, ..., *argn;

execvp(name, args)
char *name, **args;
```

DESCRIPTION

All of these `exec` routines provide occasionally convenient interfaces to the `execve` system call. In all of them, the **name** argument gives the filename, including the path, of the program to load. The `execl` family allows the argument list to be passed as a NULL-terminated list of pointers, where **arg0** is the name of the program to load. `execle` also allows a pointer to an array of pointers to strings to be passed for the environment list of the new program; the other routines all pass the current environment. `execv` differs from `execve` only in that the current environment list is passed implicitly. `execlp` and `execvp` use the value of the environment variable `PATH` to search for the program referred to by **name**. The library routine `findpath` is used to examine the path.

DIAGNOSTICS

A return from any of these routines constitutes an error. Problems arise if the requested program is not executable or accessible, if there are too many arguments, or if not enough memory is available to load the program.

SEE ALSO

System Calls - `execve()`

Library Functions - `getenv()`, `findpath()`

NAME

`exit` – terminate a process, termination

SYNOPSIS

```
exit(status)
int status;
```

DESCRIPTION

`exit` flushes all stream buffers and closes all file descriptors. It then calls `_exit`, terminating the current process.

DIAGNOSTICS

If `exit` returns at all, something has gone very wrong.

SEE ALSO

System Calls - `_exit()`

Library Functions - `fflush()`, `fclose()`

NAME

exp, log, log10,
pow, sqrt – exponential, logarithm, power, square root functions

SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;
```

DESCRIPTION

exp returns the exponential function of **x:** exp(x)= e^x

log returns the natural logarithm of **x:** log(x)= ln(x)

log10 returns the base 10 logarithm of **x:** $\log_{10}(x) = \log_{10}(x)$

pow returns the value of **x** raised to the power of **y:** pow(x,y)= x^y

sqrt returns the square root of **x:** sqrt(x)= $\sqrt{x}$

DIAGNOSTICS

log and log10 set **errno** to EDOM when **x** is non-positive. pow returns 0 and sets **errno** to EDOM when **x** is zero and **y** is non-positive, or when **x** is negative and **y** is not an integer. sqrt sets **errno** to EDOM when **x** is negative. exp and pow set **errno** to ERANGE if the return value would overflow or underflow.

SEE ALSO

Library Functions - `cbrt()`, `sinh()`

NAME

`fclose`, `fflush`,
`unlocked_fflush` - close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose(stream)
FILE *stream;

int fflush(stream)
FILE *stream;

int unlocked_fflush(stream)
FILE *stream;
```

DESCRIPTION

`fclose` writes all buffered data for the named **`stream`**, frees any dynamically-allocated buffers, and closes the file descriptor associated with the file. `exit` calls `fclose` for each open stream.

`fflush` writes all buffered data in the stream. The stream remains open.

`unlocked_fflush` is identical to its thread-safe brethren but performs no locking. If the application wishes to do its own locking, it can do so by using `flockfile` and `funlockfile`. In that case, `unlocked_fflush` will be somewhat faster.

RETURN VALUE

`fclose` and `fflush` return zero if successful, `EOF` otherwise.

SEE ALSO

System Calls - `close()`

Library Functions - `exit()`

NAME

ferror, feof,
clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int feof(stream)
FILE *stream;

int ferror(stream)
FILE *stream;

clearerr(stream)
FILE *stream;

int fileno(stream)
FILE *stream;
```

DESCRIPTION

These functions allow inquiries about the status of standard I/O streams to be made. All are actually macros defined in **stdio.h**. feof returns a non-zero value if EOF has previously been detected while reading the named input stream, or zero if not. ferror returns a non-zero value if an I/O error has previously occurred while reading from or writing to the named stream, or zero if not. clearerr resets the error indicator and EOF indicator to zero for the named **stream**.

fileno returns the integer file descriptor for the named stream. This value can be used to make system calls concerning the file, such as fstat or fchmod.

SEE ALSO

System Calls - open(), lseek(), write

Library Functions - fopen(), fseek()

NAME

`findpath` – locate a file

SYNOPSIS

```
#include <file.h>

char *findpath(path, name, mode)
char *path, *name;
int mode;
```

DESCRIPTION

`findpath` searches through the directories named in path for a regular file whose name matches **name** and which is accessible in the manner given by **mode**. The **path** string consists of a list of pathnames, separated by colons. An empty field (i.e. "`::`", or a single colon at the left or right end of the string) is taken to mean "`.`". A terminating slash for each pathname is optional. This path format is the same as that used by the shell for the `PATH` environment variable.

The **mode** indicates which combination of `read`, `write`, and `execute` permissions are desired for the file. It is constructed as the logical-OR of any of the constants `R_OK`, `W_OK`, `X_OK`, and `F_OK`, indicating read access, write access, execute access, or simple existence.

If the **name** contains a slash, no searching is performed and the name is returned unchanged. Otherwise, `findpath` returns the first matching pathname.

NOTE

The return string is stored in a static array, and so must be copied if it is to be saved. Only regular files are examined, not directories or devices.

SEE ALSO

System Calls - `access()`

Library Functions - `getenv()`

NAME

`flockfile`,
`funlockfile` – obtain/release mutually exclusive access to a stream

SYNOPSIS

```
#include <stdio.h>

void flockfile(stream)
FILE *stream;

void funlockfile(stream)
FILE *stream;
```

DESCRIPTION

`flockfile` and `funlockfile` obtain and release a mutually-exclusive lock on the stream referred to by **`stream`**. If one thread has such a lock and another thread tries to acquire the lock, the second thread will be blocked until the first thread releases the lock. If a thread has a lock and the same thread attempts to acquire the same lock again, the thread will, on the other hand, not be blocked. In other words, the locks applied to a stream `stack`.

By default, the functions that operate of streams (`printf`, `fscanf`, `fread` and so forth) all implicitly lock and unlock the stream on which they operate, so `flockfile` and `funlockfile` need not be used. However, there are `unlocked` versions of each function (`unlocked_printf`, `unlocked_scanf`, `unlocked_fread` and so forth) which do not perform implicit locking. These functions may be called safely when a thread has a stream locked by using `flockfile`. Thus, if numerous operations on the same must be performed atomically, the application can lock the stream with `flockfile`, perform the operations using the `unlocked` variants, and then unlock the stream using `funlockfile`. This may prove more efficient as well as being the correct thing to do in many cases.

NOTE

This locking only applies to streams. I/O performed using `read`, `write`, `lseek`, and so forth, will bypass the locking scheme.

RETURN

These routines do not return anything.

SEE ALSO

Library Functions - `fopen()`, `fclose()`, `fread()`, `fwrite()`, `unlocked_fread()`, `unlocked_fwrite()`, `printf()`, `fprintf()`, `unlocked_printf()`, `unlocked_fprintf()`, `scanf()`, `fscanf()`, `unlocked_scanf()`, `unlocked_fscanf()`, `getc()`, `fgetc()`, `getchar()`, `getw()`, `unlocked_getc()`, `unlocked_fgetc()`, `unlocked_getchar()`, `unlocked_getw()`, `putc()`, `fputc()`, `putchar()`, `putw()`, `unlocked_putc()`, `unlocked_fputc()`, `unlocked_putchar()`, `unlocked_putw()`

NAME

`floor`, `ceil`,
`fmod`, `fabs` – floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fmod(x, y)
double x, y;

double fabs(x)
double x;
```

DESCRIPTION

`floor` returns the largest integer no greater than **x**.

`ceil` returns the smallest integer no less than **x**.

`fmod` returns the floating-point remainder of the division of **x** by **y**. If **y** is zero or if **x/y** would overflow, `fmod` returns zero.

`fabs` returns the absolute value **|x|**.

SEE ALSO

Library Functions - `abs()`

NAME

`fopen, freopen, fdopen` – open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(name, type)
char *file_name, *type;

FILE *freopen(name, type, stream)
char *file_name, *type;
FILE *stream;

FILE *fdopen(fd, type)
int fd;
char *type;
```

DESCRIPTION

`fopen` opens the file **name** and associates a stream with it. `fopen` returns a pointer to the `FILE` structure associated with the stream.

The **name** argument points to a character string that contains the name of the file to be opened. The **type** argument points to a character string with one of the following values:

r	open for reading
w	truncate or create for writing
a	append; open or create for writing at end-of-file
r+	open for update (reading and writing)
w+	truncate or create for update
a+	random access open for read or write; pointer will be repositioned to end-of-file for writing

`freopen` substitutes the named file in place of the open stream. The original stream is closed whether or not the open succeeds. `freopen` returns a pointer to the `FILE` structure associated with stream. `freopen` is typically used to attach the pre-opened streams associated with `stdin`, `stdout`, and `stderr` to other files.

`fdopen` creates a `FILE` block for an already open file descriptor. The address of the new block is returned. There is no way that `fdopen` can verify that the type requested corresponds to the mode of the open file, so a mismatch will manifest itself as errors in future operations.

If a file is open for update, both input or output may be attempted on the stream. However, output may not be directly followed by input without an intervening `fseek` or `rewind`, and input may not be directly followed by output without an intervening `fseek`, `rewind`, or an input operation which encounters end-of-file.

In a file open for append, no information can be overwritten. All output is appended to the end of the file regardless of current pointer position. After output is completed, the pointer is positioned at the end of the file.

DIAGNOSTICS

If unsuccessful, both routines return a NULL pointer.

SEE ALSO

System Calls - `open()`

Library Functions - `fseek(), rewind()`

NAME

`fpgetround`, `fpsetround`,
`fpgetmask`, `fpsetmask`,
`fpgetsticky`, `fpsetsticky` – IEEEfloating-pointenvironmentcontrol

SYNOPSIS

```
#include <ieeefp.h>

fp_rnd fpgetround();

fp_rnd fpsetround(rnd_dir);
fp_rnd rnd_dir;

fp_except fpgetmask();

fp_except fpsetmask(mask);
fp_except mask;

fp_except fpgetsticky();

fp_except fpsetsticky(sticky);
fp_except sticky;
```

DESCRIPTION

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE Standard 754-1985) specifies that four floating-point rounding modes shall be provided. The `fpgetround()` and `fpsetround()` functions are used to get and set the rounding mode, respectively, using the following constants (defined in `<ieeefp.h>`):

`FP_RN`	round to nearest (tie -> even)
`FP_RM`	round toward minus infinity
`FP_RP`	round toward plus infinity
`FP_RZ`	round toward zero (truncate)

The default rounding mode is `FP_RN` (round to nearest).

The IEEE Standard 754-1985 specifies five types of floating-point exceptions: invalid operation, division by zero, overflow, underflow, and inexact. When one of these exceptions occurs, the corresponding sticky bit is set, and, if the corresponding enable bit is set, a trap occurs. When a trap occurs, the current thread is sent a `SIGFPE` signal. The `fpgetsticky()` and `fpsetsticky()` functions are used to get and set the sticky bits, respectively, and the `fpgetmask()` and `fpsetmask()` functions are used to get and set the exception enable bits, respectively.

The value passed to `fpsetsticky()` and the value passed to `fpsetmask()` is encoded as the bitwise-OR of zero or more of the following constants (defined in `<ieeefp.h>`):

`FP_X_INV`	invalid operation exception
`FP_X_DZ`	division by zero exception
`FP_X_OFL`	overflow exception
`FP_X_UFL`	underflow exception
`FP_X_IMP`	inexact exception

Thus, `fpsetsticky()` modifies all sticky bits simultaneously, and `fpsetmask()` modifies all enable bits simultaneously. The values returned by `fpgetsticky()`, `fpsetsticky()`, `fpgetmask()`, and `fpsetmask()` are all encoded using this same set of bitwise constants.

The default exception enable mask is 0 (all traps disabled).

NOTE

In addition to modifying the exception enable bits, `fpsetmask()` clears the sticky bit corresponding to any exception being enabled.

The ANSI C Standard requires truncation (round toward zero) for floating-point to integer conversions. This is enforced by the C compiler, and is not affected by modifying the rounding mode with `fpsetround()`.

When a trap occurs, it is necessary to clear the corresponding sticky bit before proceeding. Otherwise, when the next trap occurs, a wrong exception type may be signaled.

The FPU on the Intel x86 platform provides an exception for denormalized operand. This exception has a sticky bit and an enable bit, both represented by the following constant (defined in `<ieeefp.h>`):

`FP_X_DNML`	denormalized operand exception

The FPU on the PowerPC platform does not provide a sticky bit for the invalid operation exception. Instead, it provides a sticky bit for each type of invalid operation, represented by the following constants (defined in `<ieeefp.h>`):

`FP_X_VXSNAN`	signaling NaN
`FP_X_VXISI`	magnitude subtraction of infinities
`FP_X_VXIDI`	division of infinity by infinity
`FP_X_VXZDZ`	division of zero by zero
`FP_X_VXIMZ`	multiplication of infinity by zero
`FP_X_VXVC`	ordered comparison involving a NaN
`FP_X_VXSOFT`	software-generated
`FP_X_VXSQRT`	square root of a negative number
`FP_X_VXCVI`	invalid integer conversion

The exception enable bit for invalid operation is not subdivided like this, so the above constants are valid only for `fpgetsticky()` and `fpsetsticky()` on the PowerPC.

DIAGNOSTICS

`fpgetround()` returns the current rounding mode.

`fpsetround()` sets the rounding mode and returns the previous rounding mode.

`fpgetsticky()` returns the current exception sticky bits.

`fpsetsticky()` modifies all exception sticky bits and returns the previous setting.

`fpgetmask()` returns the current exception enable bits.

`fpsetmask()` modifies all exception enable bits and returns the previous setting.

SEE ALSO

Library Functions - `isnan()`

NAME

`fread`, `fwrite`,
`unlocked_fread`,
`unlocked_fwrite` – block input/output

SYNOPSIS

```
#include <stdio.h>

int fread(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int unlocked_fread(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int unlocked_fwrite(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

DESCRIPTION

`fread` places into an array pointed to by **ptr nitems** of data read from the input stream beginning at **ptr**. The data items are sequences of bytes of length **size**. Reading stops when an error occurs, end-of-file is encountered, or **nitems** of data have been read. `fread` places the file pointer (if applicable to the nature of the file) at the byte following the last byte read, if one exists. The contents of the stream are not changed.

`fwrite` attempts to append **nitems** of data from the array pointed to by **ptr** to the named output stream. Each data item is as described above.

`fread` and `fwrite` obtain mutually-exclusive access to **`stream`**, and are therefore thread-safe. `unlocked_fread` and `unlocked_fwrite` are identical to their thread-safe brethren but perform no locking. If the application wishes to do its own locking, it can do so by using `flockfile` and `funlockfile`.

RETURN VALUE

All routines return the number of items written or read. If a non-positive value for **`nitems`** is passed, zero is returned and nothing is read or written.

SEE ALSO

System Calls -	`lseek()`
Library Functions -	`fopen(), fclose(), flockfile(), funlockfile()`

NAME

`frexp, ldexp, modf` – split into mantissa and exponent

SYNOPSIS

```
#include <math.h>

double frexp(value, eptr)
double value;
int *eptr;

double ldexp(x, n)
double x;
int n;

double modf(value, iptr)
double value, *iptr;
```

DESCRIPTION

`frexp` returns the mantissa of **`value`** and stores the associated integer exponent at the location pointed to by **`eptr`**. The mantissa, *x*, and exponent, n, are quantities such that $0.5 \leq |x| < 1.0$ and **`value =`** $x * 2^n$.

`ldexp` returns the quantity $\mathbf{x} * 2^{\mathbf{n}}$.

`modf` returns the positive fractional part of **`value`** and stores the integral part at the location pointed to by **`iptr`**.

SEE ALSO

Library Functions - `floor()`

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek(stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind(stream)
FILE *stream;

long ftell(stream)
FILE *stream;
```

DESCRIPTION

`fseek` sets the position for the next input or output operation on the stream. The new position, taken from **offset**, is a signed number of bytes from the beginning, from the current position, or from the end of the file, depending on the value of **ptrname** (0, 1, or 2 respectively).

`rewind` is equivalent to `fseek`(**stream, 0L, 0**), except no value is returned.

`ftell` returns the **offset** of the current byte relative to the beginning of the file associated with the named stream.

`fseek` and `rewind` undo the effects of `ungetc`.

After `fseek` or `rewind`, the next operation to the file may be either input or output.

`fseek`, `rewind`, and `ftell` obtain mutually-exclusive access to stream, and are therefore thread-safe.

DIAGNOSTICS

If unsuccessful, `fseek` returns a non-zero value, otherwise zero.

SEE ALSO

System Calls - `lseek()`

Library Functions - `flockfile()`, `funlockfile()`

NAME

`fsetpos, fgetpos` – reposition a file pointer in a standard I/O stream

SYNOPSIS

```
#include <stdio.h>

int fsetpos(stream, pos)
FILE *stream;
const fpos_t *pos;

int fgetpos(stream, pos)
FILE *stream;
fpos_t *pos;
```

DESCRIPTION

The `fsetpos` function sets the file position indicator for the stream pointed to by **`stream`** according to the value of the object pointed to by **`pos`**, which must be a value obtained from an earlier call to `fgetpos` on the same stream.

A successful call to `fsetpos` clears the end-of-file indicator for the stream and undoes any effects of the `ungetc` function on the same stream. After a `fsetpos` call, the next operation on an update stream may be either input or output.

The `fgetpos` function stores the current value of the file position indicator for **`stream`** in the object pointed to by **`pos`**. The value stored contains information usable by `fsetpos` for repositioning the stream to its position at the time of the call to `fgetpos`.

DIAGNOSTICS

Both `fsetpos` and `fgetpos` return zero if successful, `EOF` otherwise.

SEE ALSO

Library Functions - `fseek(), ftell(), ungetc()`

System Calls - `lseek()`

NAME

`ftok` – derive interprocess communication key from file name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(path, id)
char *path
char id;
```

DESCRIPTION

`ftok` returns a key based on **`path`** and **`id`** that can be used in later calls to `msgget`, `semget`, and `shmget`. **`path`** must specify an existing file which the process can access, and **`id`** can be any character - usually one which uniquely identifies an application. The key is derived by combining the specified **`id`** with the device id and inode number of the file specified by **`path`**.

`ftok` will return the same key for two hard-linked files (i.e. files linked using the `link` system call) when called with the same **`id`**, because the two hard-linked files share the same inode number.

DIAGNOSTICS

`ftok` calls `stat` to get the device id and inode number of the file specified by **`path`**. If the call to `stat` fails, `ftok` will return `-1`.

NOTE

Due to the fact that device ids and inode numbers contain more bits of information than is used by `ftok`, there is no guarantee that `ftok` will return a unique key for each file, although the chance of collision is still very small. Specifically, a unique key is guaranteed only when the number of inodes in a file system does not exceed 65536, and when the device id of a file system does not exceed 256. Otherwise, a unique key is not guaranteed.

SEE ALSO

System Calls - `link()`, `msgget()`, `semget()`, `shmget()`, `stat()`

Utility Programs - `mkfs`

NAME

ftw – walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw(path, fn, depth)
char *path;

int(*fn)();
int depth;
```

DESCRIPTION

ftw recursively descends the directory hierarchy with its root at **path**. For each file in the hierarchy, ftw calls **fn**, passing it three arguments—a pointer to a null-terminated character string containing the name of the file, a pointer to a stat structure containing information about the file, and an integer. The integer can take on the following values, as defined in **ftw.h**:

FTW_F	The file is not a directory.
FTW_D	The file is a directory.
FTW_DNR	The file is a directory that cannot be read.
FTW_NS	The stat system call fails on the file.

If the integer is FTW_DNR, the descendants of that directory will not be processed. If the integer is FTW_NS, the stat structure will be invalid. A file that would cause FTW_NS to be passed to **fn** would be a file in a directory where the directory has read but not execute permission.

The ftw function visits a directory before visiting any of its descendants.

The tree traversal continues until either the tree is exhausted, **fn** returns a non-zero value, or some error is detected within ftw. If the tree is exhausted, ftw returns zero. If **fn** returns a non-zero value, ftw stops its tree traversal and returns whatever value was returned by **fn**. If ftw detects an error, it returns -1 and sets **errno** to indicate the error.

The `ftw` function uses one file descriptor for each level in the tree. The **`depth`** argument limits the number of file descriptors so used. A **`depth`** value less than 1 is treated as 1. **`depth`** must not be greater than the number of file descriptors currently available for use. `ftw` will run more quickly if **`depth`** is at least as large as the number of levels in the tree.

NOTE

Because `ftw` is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

SEE ALSO

System Calls - `stat()`

NAME

`gamma` – natural log of gamma function

SYNOPSIS

```
#include <math.h>

extern int signgam;

double gamma(x)
double x;
```

DESCRIPTION

`gamma` returns `ln` Γ**|x|**. The sign of Γ**|x|** is returned in the integer **signgam** as either positive or negative 1. Note that for positive **x**, **signgam** will always be 1.

This function returns the natural log of Γ**|x|** to provide results for larger values of **x**. Values of **x** greater than about 171 produce results from `gamma` too large for `exp`, assuming 64-bit double precision floating point.

DIAGNOSTICS

`gamma` returns large values and sets **errno** to `EDOM` for integer values of **x** less than or equal to zero. Positive error detection requires that **errno** be cleared before function calls and then checked afterwards.

SEE ALSO

Library Functions - `exp()`

NAME

`getattr, fgetattr` – get attributes of a realtime file

SYNOPSIS

```
#include <types.h> #include <files.h>

int getattr(path, actattr)
char *path;
struct attrbuf *actattr;

int fgetattr(fildes, actattr)
int fildes;
struct attrbuf *actattr;
```

DESCRIPTION

`getattr()` populates the **attrbuf** structure pointed to by **actattr** with the attributes of the realtime file named in **path**. The fields of the **attrbuf** structure match those used when the file was created. The attributes are persistent across open instances of the file.

`fgetattr()` performs the same operation as `getattr`, except that the file is specified by the open file descriptor **fildes** instead of by name.

DIAGNOSTICS

These functions return 0 if successful, or -1 if not. If `getattr` fails, **errno** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	The file does not exist.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	A directory in the path denied search permission.
EFAULT	**path** does not point to allocated memory in the process' address space.

`EINVAL`	The file is not a realtime file.

If `fgetattr` fails, **`errno`** will contain one of the following:

`EBADF`	**`fildes`** is not a valid file descriptor.
`EINVAL`	**`fildes`** does not refer to a realtime file.

CONFORMANCE

`getattr` and `fgetattr` are implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Library Functions -	`rtcreate(), setattr()`
Examples -	`getattr.c`

NAME

`getc, getchar, fgetc,`
`getw, unlocked_getc,`
`unlocked_getchar, unlocked_fgetc,`
`unlocked_getw` – get a character or word from a stream

SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar ()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;

int unlocked_getc(stream)
FILE *stream;

int unlocked_getchar ()

int unlocked_fgetc(stream)
FILE *stream;

int unlocked_getw(stream)
FILE *stream;
```

DESCRIPTION

`getc` returns the next byte from the named input stream and positions the pointer ahead one byte. `getc` is implemented as a macro.

`getchar` returns the next character from the standard input stream. `getchar` is implemented as a macro.

`fgetc` performs the same function as `getc`, however it is a true function. It is slower, but takes less space per invocation.

getw returns the next word (int or unsigned) from the named input stream.

getc, fgetc, getw, and getchar obtain mutually-exclusive access to the relevant stream, and are therefore thread-safe. The unlocked variants of these functions are identical to their thread-safe brethren but perform no locking. If the application wishes to do its own locking, it can do so by using flockfile and funlockfile. In that case, the unlocked versions will be somewhat faster.

EOF is returned if end-of-file or error is encountered. Since EOF is a valid **int**, feof or ferror will need to be used to check the success of getw. The file pointer is positioned at the next word. No special alignment is assumed.

DIAGNOSTICS

These routines return EOF when end-of-file or error is encountered.

SEE ALSO

Library Functions - fopen(), fclose(), flockfile(), funlockfile(), ungetc(), getes(), putc()

NAME

getcap, fgetcap,
fsgetcap – get capabilities of real-time files

SYNOPSIS

```
#include <types.h>
#include <files.h>

int getcap(path, reqattr, reqwhich, capbufp, capwhich)
char *path;
struct attrbuf *reqattr;
int reqwhich;
struct capbuf *capbufp;
int *capwhich;

int fgetcap(fildes, reqattr, reqwhich, capbufp, capwhich)
int fildes;
struct attrbuf *reqattr;
int reqwhich;
struct capbuf *capbufp;
int *capwhich;

int fsgetcap(path, reqattr, reqwhich, capbufp, capwhich)
char *path;
struct attrbuf *reqattr;
int reqwhich;
struct capbuf *capbufp;
int *capwhich;
```

DESCRIPTION

The `getcap()` function allows the calling process to obtain the capabilities of the realtime file specified by **path**.

The `fgetcap()` function allows the calling process to obtain the capabilities of the realtime file specified by the file descriptor **fildes**.

The `fsgetcap()` function allows the calling process to obtain the capabilities of the files to be created in a particular directory. If the **path** refers to a file which is not a directory, then the capabilities refer to the directory containing that file. If **path** refers to a directory, then the capabilities refer to the directory itself.

The **reqattr** argument specifies the attributes that are considered to be constraints. The capabilities returned by the above functions will be based on these constraints.

The **reqwhich** argument specifies which attributes in **reqattr** are to be used as constraints. Upon successful completion of the function, the **capwhich** argument will contain the attributes in **reqattr** for which the constraints cannot be met.

The **capbufp** argument points to a **capbuf** capability structure, which is declared in **files.h**.

The above functions return the value zero to the calling process if the function is successful; otherwise the function returns the value -1 and sets **errno** to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `getcap()` and `fsgetcap()` functions return -1 and sets **errno** to the corresponding value:

`EACESS`	Search permission is denied for a component of the **path** prefix.
`ENOENT`	The named file does not exist or the **path** argument points to an empty string.
`ENOTDIR`	A component of the **path** prefix is not a directory.

If any of the following conditions occur, the `fgetcap()` function returns -1 and sets **errno** to the corresponding value:

`EBADF`	The **fildes** argument is not a valid file descriptor.

CONFORMANCE

`getcap`, `fgetcap`, `fsgetcap` are implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Examples - `getcap.c`, `fsgetcap.c`

NAME

getclock,setclock,
resclock – system-wide clock handling routines.

SYNOPSIS

```
#include <sys/timers.h>

int getclock clock_type, tp)
int clock_type;
struct timespec *tp;

int setclock(clock_type, tp)
int clock_type;
struct timespec *tp;

int resclock(clock_type, res, maxval)
int clock_type;
struct timespec *res;
struct timespec *maxval;
```

DESCRIPTION

setclock is used to set the system-wide clock specified by **clock_type** to the time value **tp**. **tp** is a pointer to a **timespec** structure, which is described as follows:

```
struct timespec {
  unsigned long tv_sec;  /* seconds */
  long tv_nsec;  /* nanoseconds */
}
```

The time interval represented by this structure is
tv_sec * 1,000,000,000 + tv_nsec nanoseconds.

getclock returns the current value of the specified system-wide clock.

resclock returns the resolution and maximum value of the clock. The resolution of the clock is the time between ticks of the clock, and is usually one-one hundredth of a second. The maximum value max is the maximum possible time.

Currently, the only available system-wide clock is the `TIMEOFDAY` clock. The `TIMEOFDAY` clock represents the amount of time since January 1, 1970, 12 midnight.

DIAGNOSTICS

`getclock`, `setclock`, and `resclock` return 0 if successful. A return value of -1 indicates that an error occurred, and **`errno`** is set to one of the following values:

`EINVAL`	**`clock_type`** is not `TIMEOFDAY`.
`EINVAL`	**`tp`** is outside the range for the given **`clock_type`**.
`EINVAL`	**`tp`** specifies a nanosecond value less than zero or greater than or equal to 1000 million.
`EPERM`	The caller is not privileged (**`setclock`** only).

CONFORMANCE

`getclock, setclock`, and `resclock` are implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Library Functions - `gettimer(), time(), ctime()`

Example - `getclock.c, setclock.c, resclock.c`

NAME

`getcwd` – get pathname of current working directory

SYNOPSIS

```
char *getcwd(buf, size)
char *buf;
int size;
```

DESCRIPTION

`getcwd` computes the pathname of the current working directory of the calling process and returns a pointer to it.

`size` must be at least one greater than the length of the pathname to be returned.

`getcwd` does not check whether **`buf`** is `NULL`.

`getcwd` is implemented like the LynxOS `getwd` function—by examining the "`..`" directory entries until the root is encountered.

DIAGNOSTICS

`getcwd` returns **`buf`** if successful, or a NULL pointer if not. If `getcwd` fails, **`errno`** will contain one of the following:

`EINVAL`	**`size`** is less than or equal to zero.
`ERANGE`	**`size`** is greater than zero but smaller than the length of the pathname plus one.
`EACCES`	A directory in the path denies read or search permission.

SEE ALSO

Utility Programs - `pwd`

System Calls - `stat()`

Library Functions - `getwd()`, `malloc()`

NAME

`getenv` – retrieve environment data

SYNOPSIS

```
char *getenv(name)
char *name;
```

DESCRIPTION

`getenv` searches the list of environment variables for a string of the form **name** = `value`. If such a string is found, `getenv` returns a pointer to `value`, otherwise NULL. Note that the value resides in space allocated when the process was started; therefore, it should only be overwritten with caution.

SEE ALSO

System Calls - `execve()`

NAME

`getfsent`, `getfsspec`,
`getfsfile`, `getfstype`,
`setfsent`, `endfsent` – retrieve entries from the file system table

SYNOPSIS

```
#include <fstab.h>

struct fstab *getfsent()

struct fstab *getfsspec(spec)
char *spec;

struct fstab *getfsfile(file)
char *file;

struct fstab *getfstype(type)
char *type;

setfsent()

endfsent()
```

DESCRIPTION

Each of the routines `getfsent`, `getfsspec`, `getfsfile`, and `getfstype` return data from the file system table (`/etc/fstab`) in the form of a structure:

```
#define FSTAB_RW    "rw"    /* Read and write */
#define FSTAB_RO    "ro"    /* Read only */
#define FSTAB_XX    "xx"    /* Ignore this entry */

struct  fstab{
  char    *fs_spec;       /* Special file name */
  char    *fs_file;       /* Mount point file name */
  char    *fs_type;       /* File system type */
  int     fs_freq;        /* Not used */
  int     fs_passno;      /* Not used */
};
```

The fields of the structure contain the name of the device containing the file system, the name of the directory where the file system is normally mounted, the type of the file system (read/write or read only), and two ignored fields included for compatibility with other operating systems. The first call to `getfsent` returns the first entry in the table. Subsequent calls return entries until end of file is reached, at which point `getfsent` returns a NULL pointer. `getfsspec`, `getfsfile`, and `getfstype` first rewind the file, then search for an entry whose special file name, file system file name, or file system type matches the argument string. These routines all return a NULL pointer if no entry is found. `setfsent` rewinds the file, and `endfsent` closes the file.

These routines are used by the `mount` and `umount` programs.

NOTE

The return value points to a static structure. The contents should be copied if they are to be retained after a subsequent call.

SEE ALSO

Utility Programs - `mount, umount`

System Calls - `mount(), umount()`

NAME

getgrent, getgrnam,
getgrgid, setgrent,
endgrent, fgetgrent,
getgrent_r, getgrnam_r,
getgrgid_r, setgrent_r,
endgrent_r – get entries from group database

SYNOPSIS

```
#include <grp.h>

struct group *getgrent()

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(nam)
char *nam;

setgrent()

endgrent()

struct group *fgetgrent(f)
FILE *f;

struct group *getgrent_r(grp,result,buffer,buflen, isopen)
rlb *grp;

struct group *result;
char *buffer;
int buflen;
int *isopen;

struct group *getgrgid_r(result,gid,buffer,buflen)

struct group *result;
int gid;
char *buffer;
int buflen;

struct group *getgrnam_r(result,name,buffer,buflen)
```

```
struct group *result;
char *name;
char *buffer;
int buflen;

setgrent_r(grp, isopen)
rlb *grp;
int *isopen;

endgrent_r(grp, isopen)
rlb *grp;
int *isopen;
```

DESCRIPTION

The routines getgrent, getgrgid, and getgrnam retrieve lines in the database which records user groups, /etc/group. All return pointers to **group** structures, defined in **grp.h**:

```
struct group {
  char *gr_name;   /* Group name */
  char *gr_passwd; /* Useless and strange;
       for compatibility only */
  int gr_gid;      /* Group id number */
  char **gr_mem;   /* List of member user names */
};
```

The field gr_name is the name of the group, gr_gid is the group number, and gr_mem is a null-terminated array of strings containing names of users who are members of the group. Every user is a member of at least one group, by way of the group ID field in the passwd database. The **group** file allows individual users to be included in several groups for purposes of access rights verification.

getgrent returns the next entry in the file, opening the file if it is not open already. getgrgid returns the first entry in the file whose gr_gid field matches **gid**. getgrnam returns the first entry in the file whose gr_nam field matches **nam**.

setgrent rewinds the file; endgrent closes the file.

fgetgrent returns a pointer to the next **group** structure in the stream **f**, which matches the format of /etc/group.

`getgrgid_r`, `getgrnam_r`, `getgrent_r`, `setgrent_r`, and `endgrent_r` are thread-safe versions of the functions described above. These versions use no static storage and may therefore be called by multiple threads at once.

NOTE

The return values of the non-thread reentrant versions point to static structures, so the contents should be copied if they are to be saved after subsequent calls.

SEE ALSO

Utility Programs - `groups`

System Calls - `access()`, `open()`, `getgid()`

Library Functions - `buildgroups()`, `getpwent()`

NAME

`getgroups` – get supplementary group IDs (POSIX-compatible)

SYNOPSIS

```
getgroups(gcount, glist)
int gcount;
int glist[];
```

DESCRIPTION

In addition to the group ID associated with each user ID in the password file (`/etc/passwd`), a set of groups can be assigned to each user ID through the group file, `/etc/group`. Each system group is given a name and a list of member user IDs in this file. The group IDs are put into the array **`glist`**. The number of elements in **`glist`** is passed as **`gcount`**. `getgroups` returns the current group list in **`glist`**. The number of available entries in the array is passed as the integer **`gcount`**. No more than `NGROUPS` groups will ever be returned. If **`gcount`** is 0, `getgroups` will return the number of supplementary group IDs, but will not change the contents of the array **`glist`**.

NOTE

In order to use the POSIX-compatible `getgroups`, rather than the `getgroups` system call, you must link your program with the POSIX library (`/lib/libc_p.a`).

DIAGNOSTICS

If `getgroups` succeeds, it returns the number of groups. The actual list of group IDs will be in **`glist`**. Otherwise, `getgroups` returns -1 and places the following value in **`errno`**:

`EINVAL`	**`gcount`** is less than the number of supplementary group IDs and is not 0.

SEE ALSO

Utility Programs - `groups`

System Calls - `setregid(), getregid()`

NAME

gethostent,
gethostbyaddr,
gethostbyname,
sethostent,
endhostent – get network host entry

SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostent()

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr;
int len, type;

sethostent(stayopen)
int stayopen;

endhostent()
```

DESCRIPTION

gethostent, gethostbyname, and gethostbyaddr each return a pointer to an object with the following structure containing the broken-out fields of a line in the network host data base, /etc/hosts.

```
struct hostent {
 char  *h_name;  /* official name of host */
 char  **h_aliases;  /* alias list */
 int  h_addrtype;    /* address type */
 int  h_length;      /* length of address */
 char  *h_addr;      /* address */
};
```

The members of this structure are:

h_name   Official name of the host.

`h_aliases`	A zero terminated array of alternate names for the host.
`h_addrtype`	The type of address being returned; currently always `AF_INET`.
`h_length`	The length, in bytes, of the address.
`h_addr`	A pointer to the network address for the host. Host addresses are returned in network byte order.

`gethostent` reads the next line of the file, opening the file if necessary.

`sethostent` opens and rewinds the file. If the **`stayopen`** flag is non-zero, the host data base will not be closed after each call to `gethostent` (either directly, or indirectly through one of the other `gethost` calls).

`endhostent` closes the file.

`gethostbyname` and `gethostbyaddr` sequentially search from the beginning of the file until a matching host name or host address is found, or until EOF is encountered. Host addresses are supplied in network order.

NOTE

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

DIAGNOSTICS

Null pointer (0) returned on `EOF` or error.

FILES

`/etc/hosts`

SEE ALSO

File Formats - `hosts`

NAME

`gethz` – return the frequency of the system clock in ticks per second

SYNOPSIS

```
int gethz()
```

DESCRIPTION

`gethz` returns the frequency of the system clock in ticks per second.

NOTE

For complete compatibility with System V UNIX, `gethz` searches for the environment variable HZ and returns its value.

If HZ is not defined in the environment, or if the value of HZ cannot be interpreted as a numeric integer, `gethz` returns 0.

NAME

getincr, fgetincr,
fsgetincr – get increment lists

SYNOPSIS

```
#include <types.h>
#include <files.h>

int getincr(path, incr, capbufp, buffer, buflen)
char *path;
int incr;
struct capbuf *capbufp;
off_t *buffer;
int buflen;

int fgetincr(fildes, incr, capbufp, buffer, buflen)
int fildes;
int incr;
struct capbuf *capbufp;
off_t *buffer;
int buflen;

int fsgetincr(path, incr, capbufp, buffer, buflen)
char *path;
int incr;
struct capbuf *capbufp;
off_t *buffer;
int buflen;
```

DESCRIPTION

The `getincr()` function allows the calling process to obtain an increment list for a capability attribute for the file specified by **path**.

The `fgetincr()` function allows the calling process to obtain an increment list for a capability attribute for a file specified by **fildes**.

The `fsgetincr()` function allows the calling process to obtain the increment list for a capability attribute of files to be created in a particular directory. If **path** refers to a file which is not a directory, then the capabilities refer to the directory containing that file. If **path** refers to a directory, then the capabilities refer to the directory itself.

The **incr** argument specifies which increment is required:

ATC_ALLOCINCR	Preallocation increment.
ATC_EXTINCR	Extent size increment.
ATC_BIOINCR	Suggested transfer size increment.
ATC_ATINCR	Atomic transfer size increment.
ATC_DIOINCR	Direct I/O transfer size increment.
ATC_DATINCR	Atomic transfer size for Direct I/O increment.

The **capbufp** argument specifies the capability attributes associated with the increment list.

The **buffer** argument points to a list of type `off_t` that will receive the list.

The **buflen** argument indicates the maximum number of list entries that may be returned.

Upon successful completion, the above functions returns the number of entries stored in the buffer; otherwise the functions return -1 and set **errno** to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the `getincr()` and `fsgetinr()` function returns -1 and sets **errno** to the corresponding value:

EACCES	Search permission is denied for a component of the path prefix.
EINVAL	The specified capability does not have an increment list, or an invalid increment list was specified in the **incr** argument, or the get capabilities function is not supported for the specified file.

`ENOENT`	The named file does not exist or the **`path`** argument points to an empty string.
`ENOTDIR`	A component of the path prefix is not a directory.

If any of the following conditions occur, the `fgetincr()` function returns -1 and sets **`errno`** to the corresponding value:

`EBADF`	The **`fildes`** argument is not a valid file descriptor.

NAME

getlogin, getlogin_r – get user name given at login time

SYNOPSIS

```
char *getlogin()

int getlogin_r(name, namelen)
char *name;
int namelen;
```

DESCRIPTION

getlogin returns the name used by the owner of the calling process when the user first logged in to the shell. The login utility associates (in the /etc/utmp file) the user name it is given with the terminal line it is using. The getlogin routine should be used as a first attempt to determine the correct current user name, in case several user names map to the same user ID.

If there is no current control terminal, or if the slot in utmp is empty (an anomalous situation), getlogin returns NULL. Thus the correct method for determining the current user name is as follows:

```
struct pwd *p;
char *n;
if (!((n = getlogin()) && (p = getpwnam(n))) || |
!(p = getpwuid(getuid())))
  nam = 0;
else
  nam = p->pw_name;
```

getlogin_r is a thread-safe version of getlogin that stores the retrieved login name in the passed buffer, rather than an internal static area.

NOTE

The return values of the non-thread reentrant versions point to static structures, so the contents should be copied if they are to be saved after subsequent calls.

SEE ALSO

Utility Programs - `who`

System Calls - `getuid()`

Library Functions - `getpwent()`

File Formats - `passwd`

NAME

getnetent,
getnetbyaddr,
getnetbyname,
setnetent,
endnetent – get network entry

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net, type)
long net;
int type;

setnetent(stayopen)
int stayopen;

endnetent()
```

DESCRIPTION

getnetent, getnetbyname, and getnetbyaddr each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, /etc/networks.

```
struct netent {
  char    *n_name;     /* official name of net */
  char    **n_aliases; /* alias list */
  int     n_addrtype;  /* net number type */
  long    n_net;       /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

`n_aliases`	A zero terminated list of alternate names for the network.
`n_addrtype`	The type of the network number returned; currently only `AF_INET`.
`n_net`	The network number. Network numbers are returned in machine byte order.

`getnetent` reads the next line of the file, opening the file if necessary.

`setnetent` opens and rewinds the file. If the **`stayopen`** flag is non-zero, the net data base will not be closed after each call to `getnetent` (either directly, or indirectly through one of the other `getnet` calls).

`endnetent` closes the file.

`getnetbyname` and `getnetbyaddr` sequentially search from the beginning of the file until a matching net name or net address is found, or until `EOF` is encountered. Network numbers are supplied in host order.

NOTE

All information is contained in a static area so it must be copied if it is to be saved. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

DIAGNOSTICS

Null pointer (0) returned on `EOF` or error.

FILES

`/etc/networks`

SEE ALSO

File Formats - `networks`

NAME

getopt, optarg,
optind, opterr – get option letter from argument vector

SYNOPSIS

```
int getopt(argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
extern int opterr;
```

DESCRIPTION

getopt returns the next option letter in **argv** that matches a letter in **optstring**. Only **argv** entries that begin with a dash (–) and are at least two characters long can contain options letters. **optstring** should contain the option letters recognized by the program calling getopt. If a letter in **optstring** is followed by a colon, the option is expected to have an option-argument, or group of option-arguments, which may or may not be separated from the option by white space.

Upon return from getopt, optarg points to the start of the option-argument, and optind is the **argv** index of the next entry to be processed. Subsequent calls to getopt use the value in optind to determine where to start scanning the **argv** vector. optind is external and is initialized to 1 before the first call to getopt.

When the next **argv** entry to be processed is either a non-option entry (i.e., an entry not beginning with a dash (–) or shorter than two characters) or the special entry --, getopt returns -1 to indicate the end of the options. If the options were ended by a non-option entry, getopt sets optind to the **argv** index of that entry. If the options were ended by the special double-dash (--) entry, getopt sets optind to the **argv** index of the entry following the double-dash (--). In this way, the double-dash (--) can be used to indicate the end of the options.

DIAGNOSTICS

If getopt encounters either an option letter that is not in **optstring** or an option letter without an expected option-argument, it prints an error message on the standard error channel and returns a question mark (?). This error message may be disabled by setting opterr to 0.

EXAMPLE

The following program echoes each of its recognized options along with their expected option-arguments. The options -x and -y do not have any expected option-arguments, while the options -z and -w do:

```
extern char *optarg;
extern int optind;
main(argc, argv)
int argc;
char **argv;
{
  int c;
  while ((c = getopt(argc, argv, "xyz:w:")) != -1) {
    switch (c) {
    case 'x':
    case 'y':
      printf("%c\n", c);
      break;
    case 'z':
    case 'w':
      printf("%c %s\n", c, optarg);
      break;
    case '?':
      errflg++;
    }
  }
  if (errflg) {
    fprintf(stderr, "usage: %s -x -y -z arg -w arg",\
    argv[0]);
    exit(1);
  }
}
```

NAME

`getpass` – discretely request a password

SYNOPSIS

```
char *getpass(prompt)
char *prompt;
```

DESCRIPTION

`getpass` opens the current control terminal, prints **`prompt`**, and reads a response string of up to eight characters. If the control terminal (`/dev/tty`) cannot be opened, `getpass` uses standard input and output. Character echoing is disabled while the response string is read to foil the efforts of nefarious persons would watch while the password is typed. The terminating newline of the response string is replaced by a null, and is not counted in the eight character limit.

DIAGNOSTICS

`getpass` returns NULL if it cannot control the terminal.

SEE ALSO

Utility Programs - `login`

Library Functions - `crypt()`, `getlogin()`

NAME

`getpgrp` – get process group ID (POSIX-compatible)

SYNOPSIS

```
#include <sys/types.h>

pid_t getpgrp();
```

DESCRIPTION

`getpgrp` returns the process group ID of the calling process.

NOTE

The POSIX compatible version of `getpgrp` differs from the LynxOS `getpgrp` system call in that the POSIX version expects no argument. In order for a C program to use the POSIX compatible `getpgrp` instead of the LynxOS system call, it must be compiled either with `POSIXCC` set (to anything) in the environment or with the `-X` option:

```
cc -X [options ...] file.c ...
```

When cc is invoked in either of these two ways, it will produce the following line to the linker:

```
ld [options...] /lib/init.o file.o .../lib/libc_p.a \
/lib/libc.a
```

If `cc` is invoked with the `-c` option to create an object module for later linking, then the linker line must include the POSIX library (`/lib/libc_p.a`) as shown above.

SEE ALSO

System Calls - `setpgid(), setsid(), sigaction()`

NAME

getprio – get process priority

SYNOPSIS

```
#include <sys/types.h>
#include <sys/sched.h>

int getprio(pid)
pid_t pid;
```

DESCRIPTION

getprio returns the priority of the process specified by **pid**. If **pid** is 0, the priority for the calling process shall be returned.

DIAGNOSTICS

getprio returns priority value for process **pid** if successful, or –1 if not.

If getprio fails, **errno** will contain one of the following:

ESRCH **pid** does not specify a valid process.

CONFORMANCE

getprio is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - getpid(), getscheduler(), setprio(), setscheduler()

Example - getprio.c

NAME

`getprotoent`,
`getprotobynumber`,
`getprotobyname`,
`setprotoent`,
`endprotoent` – get protocol entry

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen;

endprotoent()
```

DESCRIPTION

`getprotoent`, `getprotobyname`, and `getprotobynumber` each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, `/etc/protocols`.

```
struct protoent {
  char *p_name;      /* official name of protocol */
  char **p_aliases;  /* alias list */
  long p_proto;      /* protocol number */
};
```

The members of this structure are:

`p_name`	The official name of the protocol.
`p_aliases`	A zero terminated list of alternate names for the protocol.

`p_proto` The protocol number.

`getprotoent` reads the next line of the file, opening the file if necessary.

`setprotoent` opens and rewinds the file. If the **`stayopen`** flag is non-zero, the net data base will not be closed after each call to `getprotoent` (either directly, or indirectly through one of the other `getproto` calls).

`endprotoent` closes the file.

`getprotobyname` and `getprotobynumber` sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

DIAGNOSTICS

Null pointer (0) returned on `EOF` or error.

NOTE

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

FILES

`/etc/protocols`

SEE ALSO

File Formats - `protocols`

NAME

`getpw` – get name from UID in the password file

SYNOPSIS

```
int getpw(uid, buf)
int uid;
char *buf;
```

DESCRIPTION

`getpw` searches the password file for an entry whose user ID equals **`uid`**, copies the entry into the array pointed to by **`buf`**, and returns 0. If no match is found, `getpw` returns a non-zero value.

SEE ALSO

Library Functions - `getpwent()`

File Formats - `passwd`

NAME

`getpwent,getpwuid,`
`getpwnam,setpwent,`
`endpwent,fgetpwent,`
`getpwent_r, getpwuid_r,`
`getpwnam_r, setpwent_r`
`endpwent_r-` retrieve entries from password file

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(uid)
int uid;

struct passwd *getpwnam(name)
char *name;

void setpwent()

void endpwent()

struct passwd *fgetpwent(f)
FILE *f;

struct passwd *getpwent_r(pwd, result, buffer, buflen,
isopen)
rlb *pwd;
struct passwd *result;
char *buffer;
int buflen;
int *isopen;

struct passwd *getpwuid_r(result, uid, buffer, buflen)
struct passwd *result;
int uid;
char *buffer;
int buflen;

struct passwd *getpwnam_r(result, name, buffer, buflen)
struct passwd *result;
```

```
char *name;
char *buffer;
int buflen;

void setpwent_r(pwd, isopen)
rlb *pwd;
int *isopen;

void endpwent_r(pwd, isopen)
rlb *pwd;
int *isopen;
```

DESCRIPTION

Each of the routines `getpwent`, `getpwuid`, and `getpwnam` returns a pointer to a structure defined as follows in `<pwd.h>`:

```
struct passwd {
  char    *pw_name;
  char    *pw_passwd;
  int     pw_uid;
  int     pw_gid;
  int     pw_quota;
  char    *pw_comment;
  char    *pw_gecos;
  char    *pw_dir;
  char    *pw_shell;
}
```

The fields of the structure contain the user name, the (encrypted) password, the user and group numbers, a description of the entry (in the `pw_gecos` field; the `pw_comment` field is unused, as is `pw_quota`), the login directory, and the login shell.

`getpwent` sequentially retrieves password entries, opening the password file if necessary. `setpwent` rewinds the password file, and `endpwent` closes it. Unless `endpwent` is called, the password file will remain open until the process exits or calls `execve` (the close-on-exec flag is set for the file descriptor).

`getpwuid` and `getpwnam` rewind the file and search for an entry with a matching user number or user name.

`fgetpwent` returns a pointer to the next **passwd** structure in the stream **f**, which matches the format of `/etc/passwd`.

`getpwuid_r`, `getpwnam_r`, `getpwent_r`, `setpwent_r`, and `endpwent_r` are thread-safe versions of the functions described above. These versions use no static storage and may therefore be called by multiple threads at once.

RETURN VALUE

The non-thread reentrant versions return either a pointer to a static **passwd** structure, or `NULL` if end-of-file or unsuccessful search. The thread reentrant versions return `0` is the search is successful or `-1` if unsuccessful.

SEE ALSO

File Formats - `passwd`

NOTES

The return values of the non-thread reentrant versions point to static structures, so the contents should be copied if they are to be saved after subsequent calls.

NAME

`getrpcent`, `getrpcbyname`, `getrpcbynumber`, `setrpcent`, `endrpcent` – get RPC entry

SYNOPSIS

```
#include <rpc/netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbyname(name)
char *name;

struct rpcent *getrpcbynumber(number)
int number;

setrpcent(stayopen)
int stayopen;

endrpcent()
```

DESCRIPTION

`getrpcent()`, `getrpcbyname()`, and `getrpcbynumber()` each return a pointer to an object with the following structure containing the broken-out fields of a line in the RPC program number data base, `/etc/rpc`.

```
struct rpcent {
  char *r_name;  /* name of server for this rpc program */
  char **r_aliases;  /* alias list */
  long r_number;  /* RPC program number */
};
```

The members of this structure are:

`r_name`	The name of the server for this rpc program.
`r_aliases`	A zero terminated list of alternate names for the RPC program.
`r_number`	The RPC program number for this service.

getrpcent() reads the next line of the file, opening the file if necessary.

setrpcent() opens and rewinds the file. If the **stayopen** flag is non-zero, the net data base will not be closed after each call to endrpcent() (either directly, or indirectly through one of the other getrpc calls). Once the **stayopen** flag is set to a non zero value, it cannot be zeroed.

endrpcent() closes the file if the **stayopen** flag has not been set.

getrpcbyname() and getrpcbynumber() sequentially search from the beginning of the file until a matching rpc program name or program number is found, or until end-of-file is encountered. These two functions use **setrpcent** function to rewind the file, use getrpcent function repeatedly to search for the desired entry, and then will call endrpcent function to close the file.

NOTE

All information is contained in a static area so it must be copied if it is to be saved.

FILES

/etc/rpc

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

SEE ALSO

Utility Programs - rpcinfo

Library Functions - rpc()

NAME

getrpcport – get RPC port number

SYNOPSIS

```
int getrpcport(host, prognum, versnum, proto)
char *host;
int prognum, versnum, proto;
```

DESCRIPTION

getrpcport() returns the port number for version **versnum** of the RPC program **prognum** running on **host** and using protocol **proto**. It returns 0 if it cannot contact the portmapper, or if **prognum** is not registered. If **prognum** is registered but not with version **versnum** , it will still return a port number (for some version of the program) indicating that the program is indeed registered. The version mismatch will be detected upon the first call to the service.

NAME

`gets,fgets,`
`unlocked_gets,`
`unlocked_fgets` – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;

char *unlocked_gets(s)
char *s;

char *unlocked_fgets(s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

`gets` reads characters from the standard input stream, `stdin`, into the array pointed to by **s**, until an end- of-file or new-line character is encountered. The new-line character is discarded and the string is terminated with a null character.

`fgets` reads characters from the stream into an array pointed to by **s**, until `n-1` characters are read, or a new-line character is read and transferred to **s**, or end-of-file is encountered. The string is terminated with a null character.

Both `gets` and `fgets` are safe for use by multiple threads at the same time, because they perform implicit locking of the input stream. Such locking requires a certain amount of overhead. If you use `flockfile` and `funlockfile` to manually lock and unlock I/O streams, or if you know that there will be no access by concurrent threads to the input ctream, then you can use `unlocked_gets` and `unlocked_fgets`, which do not perform any locking and which are therefore incrementally faster than the thread-safe versions.

RETURN VALUE

If successful, **s** is returned.

If end-of-file is encountered and no characters have been read, then no characters are transferred to **s** and a NULL pointer is returned. A NULL pointer is also returned if an error occurs or if these functions are applied to a file not open for reading.

SEE ALSO

Library Functions - `puts()`, `getc()`, `scanf()`, `fread()`, `ferror()`, `fopen()`, `flockfile()`, `funlockfile()`

NAME

getservent, getservbyport,
getservbyname, setservent,
endservent – get service entry

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port;
char *proto;

setservent(stayopen)
int stayopen;

endservent()
```

DESCRIPTION

getservent, getservbyname, and getservbyport each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, /etc/services.

```
struct servent {
  char *s_name;      /* official name of service */
  char **s_aliases;  /* alias list */
  int s_port;        /* port service resides at */
  char *s_proto;     /* protocol to use */
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero terminated list of alternate names for the service.

`s_port`	The port number at which the service resides. Port numbers are returned in network byte order.
`s_proto`	The name of the protocol to use when contacting the service.

`getservent` reads the next line of the file, opening the file if necessary. `setservent` opens and rewinds the file. If the **`stayopen`** flag is non-zero, the net data base will not be closed after each call to `getservent` (either directly, or indirectly through one of the other `getserv` calls).

`endservent` closes the file.

`getservbyname` and `getservbyport` sequentially search from the beginning of the file until a matching protocol name or port number is found, or until `EOF` is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

NOTE

All information is contained in a static area so it must be copied if it is to be saved. Expecting port numbers to fit in a 32 bit quantity is probably naive.

DIAGNOSTICS

Null pointer (0) returned on `EOF` or error.

FILES

`/etc/services`

SEE ALSO

Library Functions - `getprotoent()`

File Formats - `services`

NAME

`getstid` – get system thread ID

SYNOPSIS

```
#include <time.h>

#include <st.h>

tid_t getstid()
```

DESCRIPTION

`getstid` returns the system thread ID for the calling system thread. The thread ID is used as a thread identifier in most thread control system and library calls.

DIAGNOSTICS

`getstid` never fails.

NOTE

`getstid` is a macro accessing a piece of thread-specific data which is initialized to contain the thread ID when the thread is first started up. This value can be accidentally overwritten by an errant program, just as any value in the writable address space of the thread can be. However, the macro is an order of magnitude faster than the system call.

The system call version of `getstid` is `_getstid`.

SEE ALSO

System Calls - `_getstid(), st_exit(), st_join(), fork()`

Library Functions - `st_default_attr()`

NAME

`gettimer`, `reltimer`,
`abstimer`, `resrel`,
`resabs` – interval timer routines.

SYNOPSIS

```
#include <sys/timers.h>

int gettimer(timerid, value)
timer_t timerid;

struct itimerspec *value;
int abstimer(timerid, value, ovalue)
timer_t timerid;

struct itimerspec *value;

struct itimerspec *ovalue;
int reltimer(timerid, value, ovalue)
timer_t timerid;

struct itimerspec *value;

struct itimerspec *ovalue;
int resabs(timerid, res, max)
timer_t timerid;

struct timespec *res;

struct timespec *max;
int resrel(timerid, res, max)
timer_t timerid;

struct timespec *res;

struct timespec *max;
```

DESCRIPTION

Interval timer values are represented by a **struct itimerspec**, which is described as follows:

```
struct itimerspec {
  struct timespec it_value;     /* initial expiration */
  struct timespec it_interval;  /* timer period */
}
```

The **timespec** structure is described as follows:

```
struct timespec {
  unsigned long tv_sec;  /* seconds */
  long tv_nsec;          /* nanoseconds */
}
```

The time interval represented by this structure is
`tv_sec * 1,000,000,000 + tv_nsec` nanoseconds.

`gettimer` returns the current value of timer **timerid**. `it_value` contains the amount of time till the next clock expiration, or 0 if the clock is disabled. `it_interval` contains the timer period. The time values are rounded up to the clock resolution.

`reltimer` and `abstimer` set the timer expiration time and period. `reltimer` sets the timer expiration for the current time plus `it_value`. `abstimer` sets the timer expiration date to an absolute time `it_value`. This time is the amount of time since January 1, 1970, 12 midnight. If this time is already passed, the timer will expire and the process will be notified in the usual fashion.

In either case, if the value represented by the `it_value` field is 0, the timer is disabled. If the **ovalue** parameter points to a valid **itimerspec** structure, the current timer settings are copied into it.

If the value represented by the `it_interval` field is non-zero, the timer will expire at regular intervals after the initial expiration. The length of the period is specified by `it_interval`.

`resabs` and `resrel` return the resolution and maximum value of the specified timer. The resolution of the clock is the time between ticks of the clock, and is usually one-one hundredth of a second. The maximum value **max** is the maximum possible time.

DIAGNOSTICS

All functions return 0 if successful. A return value of -1 indicates that an error occurred, and **e rrno** is set to one of the following values:

EINVAL	**timerid** is not a valid timer ID.
EINVAL	A **value** specifies a nanosecond value less than zero or greater than or equal to 1000 million.

CONFORMANCE

`gettimer, reltimer, abstimer, resrel, resabs` are implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`getclock(), mktimer()`
Examples -	`abstimer.c, gettimer.c, reltimer.c`

NAME

getttyent, getttyname,
getttynam, getttyidx,
setttyent, endttyent – retrieve data from tty type file

SYNOPSIS

```
#include <ttytab.h>

struct ttytab *getttyent()

struct ttytab *getttyname(name)
char *name;

struct ttytab *getttynam(name)
char *name;

struct ttytab *getttyidx(idx)
int idx;

setttyent()

endttyent()
```

DESCRIPTION

The functions `getttyent`, `getttyname`, `getttynam`, and `getttyidx` return information from the system terminal type data file (`/etc/ttys`). The data are placed in a structure as follows:

```
struct  ttytab {
  char *tty_name;      /* name of file in /dev */
  int tty_enable;      /* is tty enabled */
  char *tty_config;    /* default configuration */
  char *tty_type;      /* initial TERM value */
  char *tty_start;     /* pathname of initial (login) appl. */
  int tty_utmpidx;     /* record number in /etc/ttys file */
};
```

The fields of the structure contain the device name of the terminal, a flag which is non-zero if the terminal is available for user login, the default configuration name, the default terminal type name, the pathname of the program used to begin terminal sessions (usually `/bin/login`), and the line number of the entry in `/etc/ttys`. The line number is used by `login` and `who` for accounting.

`getttyent` returns entries in order from the file, one per call, until end of file is reached. `getttyname` rewinds the file and returns the first entry with a matching terminal device name. `getttynam` is a synonym for `getttyname`. `getttyidx` rewinds the file and returns the entry in the given position in the file. `setttyent` rewinds the file, and `endttyent` closes the file.

SEE ALSO

Utility Programs - `tset, stty`

Library Functions - `ttyconfig(), termcap()`

NAME

getutent, getutline, pututline, setutent, endutent, utmpname – retrieve entries from utmp file

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent()

struct utmp *getutline(line)
struct utmp *line;

void pututline(utmp)
struct utmp *utmp;

void setutent()

void endutent()

void utmpname(file)
char *file;
```

DESCRIPTION

The routines getutent and getutline returns a pointer to a **utmp** structure, defined in **utmp.h** as follows:

```
struct utmp {
  char ut_line[8];    /* name of tty, minus path prefix */
  char ut_name[8];    /* user name */
  char ut_host[16];   /* host name, if from network */
  char ut_time;       /* time of entry */
}
```

getutent sequentially retrieves **utmp** entries from the **utmp** file, opening the file if necessary. The **utmp** file is /etc/utmp, by default, but may be changed using utmpname (see below). getutline search for an entry with an ut_line field that matches its argument. Both getutent and getutline return NULL if they reach the end of the file.

`pututline` writes the given **utmp** structure into the **utmp** file. `setutent` rewinds the **utmp** file, and `endutent` closes it. Unless `endutent` is called, the **utmp** file will remain open until the process exits or calls execve (the close-on-exec flag is set for the file descriptor).

`utmpname` allows the user to change the name of **utmp** file from `/etc/utmp` to any other file. `utmpname` does not open the file. It just closes the old file if it is currently open and saves the new file name.

RETURN

`getutent` and `getutline` return either a pointer to a static **utmp** structure, or NULL if end-of- file or unsuccessful search.

SEE ALSO

File Formats - `utmp`

NAME

getwd – construct pathname of current working directory

SYNOPSIS

```
#include <dir.h>

char *getwd(pathname)
char pathname[MAXPATHLEN];
```

DESCRIPTION

getwd computes the pathname of the current working directory of the calling process by examining the ".." directory entries until the root is encountered.

The pathname is placed in an array pointed to by pathname. For convenience, getwd returns the address of **pathname**. The array to which **pathname** points should be large enough to contain the largest possible pathname, as defined by the constant **MAXPATHLEN**. The system will not allocate the pathname buffer.

DIAGNOSTICS

If getwd encounters an error, it returns NULL and places an error message in **pathname**.

SEE ALSO

Utility Programs - pwd

System Calls - stat()

Library Functions - getcwd()

NAME

hsearch, hcreate,
hdestroy – manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch(item, action)
ENTRY item;
ACTION action;

int hcreate(nel)
unsigned int nel;

void hdestroy()
```

DESCRIPTION

hsearch builds and manages a hash table. **item** is a structure of type **ENTRY**, defined in **search.h** as follows:

```
typedef struct entry {
  char *key;
  char *data;
} ENTRY;
```

item.key points to the comparison key, and item.data points to any other data to be associated with that key. **action** is a member of an enumeration type **ACTION**, defined in **search.h** as follows:

```
typedef enum { FIND, ENTER } ACTION;
```

action is used only if the entry cannot be found in the table: ENTER indicates that the item should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

hcreate allocates sufficient space for the table and must be called before hsearch is used. **nel** is an estimate of the maximum number of entries that the table will contain.

`hdestroy` destroys the search table and may be followed by another call to `hcreate`.

DIAGNOSTICS

`hsearch` returns a pointer into the hash table indicating the location at which an entry can be found. Otherwise, `hsearch` returns a NULL pointer if either the action if `FIND` and the item could not be found, or the action is `ENTER` and the table is full.

NOTE

`hsearch` and `hcreate` use `malloc` to allocate space.

Only one hash table may be active at any given time.

SEE ALSO

Library Functions - `bsearch()`, `tsearch()`, `lsearch()`, `qsort()`, `malloc()`

NAME

htonl, htons,
ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

```
#include <types.h>
#include <netinet/in.h>

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order.

These routines are most often used in conjunction with Internet addresses and ports as returned by `gethostent` and `getservent`.

SEE ALSO

Library Functions - `gethostent(), getservent()`

NAME

`IEEEerr` – inspect IEEE 754 floating point value

SYNOPSIS

```
#include <fperr.h>

int IEEEerr(n)
double n;
```

DESCRIPTION

IEEEerr examines the double argument n and determines whether it is an "infinity" (positive or negative), a "not a number" ("NaN"), or a usable floating point value according to IEEE 754 specifications. If the value is not usable, the global variable **errno** is set to `ERANGE`. Additionally, `IEEEerr` returns an error code to describe the value as follows:

`FPERR_PINF`	Positive infinity
`FPERR_MINF`	Negative infinity
`FPERR_NAN`	Not a number
`FPERR_OK`	Usable value

The constants are defined in `fperr.h` such that error values are negative and the OK value is zero.

Note that `printf` examines floating point values before formatting and produces an appropriate mnemonic for unusable values.

SEE ALSO

Library Functions - `printf()`

NAME

`inet_addr`, `inet_network`,
`inet_ntoa`, `inet_makeaddr`,
`inet_lnaof`, `inet_netof` – Internet address manipulation routines

SYNOPSIS

```
#include <types.h>
#include <socket.h>
#include <bsd/in.h>
#include <arpa/inet.h>

ulong inet_addr(cp)
char *cp;

void inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

u_long inet_lnaof(in)
struct in_addr in;

u_long inet_netof(in)
struct in_addr in;
```

DESCRIPTION

The routines `inet_addr` and `inet_network` each interpret character strings representing numbers expressed in the Internet standard "." notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine `inet_ntoa` takes an Internet address and returns an ASCII string representing the address in "." notation. The routine `inet_makeaddr` takes an Internet network number and a local network address and constructs an Internet address from it. The routines `inet_netof` and `inet_lnaof` break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet address are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the “`.`” notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as `128.net.host`.

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as `net.host`.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as “parts” in a “`.`” notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e. a leading `0x` or `0X` implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The value -1 is returned by `inet_addr` and `inet_network` for malformed requests.

NOTE

The problem of host byte ordering versus network byte ordering is confusing. A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed. The string returned by `inet_ntoa` resides in a static memory area.

SEE ALSO

Library Functions -	`gethostent()`, `getnetent()`
File Formats -	`hosts`, `networks`

NAME

`initgroups` – initialize supplementary group IDs

SYNOPSIS

```
int initgroups(name, initial_gid)
char *name;
int initial_gid;
```

DESCRIPTION

`initgroups()` reads through the group file and sets up, using the `setgroups` call (see `getgroups`), the supplementary group IDs for the user specified in *name*. The `initial_gid` is automatically included in the supplementary group IDs. Typically this value is given as the group number from the password file.

FILES

`/etc/passwd`

`/etc/group`

DIAGNOSTICS

`initgroups()` returns `-1` if the supplementary group IDs could not be set.

BUGS

`initgroups()` uses the routines based on `getpwent` and `getgrent`. If the invoking program uses any of these routines, the password or group structure will be overwritten in the call to `initgroups`.

SEE ALSO

Library Functions - `getpwent()`, `getgrent()`, `getgroups()`

NAME

`iosuspend` – wait for one of a set of asynchronous requests

SYNOPSIS

```
#include <sys/aio.h>

int iosuspend(nent, aiocbps)
int nent;
struct aiocb *aiocbps[];
```

DESCRIPTION

`iosuspend` suspends execution of the calling thread until one of the asynchronous operations described by the array **aiocbps** completes, or until a signal interrupts the `iosuspend` call. The size of the array is **nent** elements; NULL entries in the array are ignored.

The first asynchronous operation to complete will cause the thread to awaken; `iosuspend` will return 0. The application should look at the `aio_errno` field for each **aiocb** to determine which I/O operation completed.

If any of the I/O operations have completed by the time of the `iosuspend` call, the call returns without suspending the calling thread.

NOTE

If the specific **aiocbp** is not NULL, and the **aiocbp** was NOT last used for the specific operation to be waited for, undesirable results may occur (such as the operation it was most recently used for being the one to be waited for).

NOTE

`iosuspend` is interruptible by signal delivery, and asynchronous requests often result in signal delivery. A signal delivery may cause `iosuspend` to fail (**errno**=`EINTR`) due to the completion of one of the waited- for operations. Suggestions to prevent this are: (1) do not have the I/O send an event on completion; (2) make sure the I/O operation is requested by a different thread than the one doing the `iosuspend` call; (3) mask signals as needed.

To use this function, include the library `libc_p.a` in the compilation command line.

DIAGNOSTICS

If `iosuspend` succeeds, it will return 0; if it fails, -1 is returned.

If the call fails, **errno** will contain one of the following:

EINTR	A signal or event was delivered.
EINVAL	All of the elements in the array **aiocbps** are NULL.
EFAULT	**aiocbp** is a not a valid address.

CONFORMANCE

`iosuspend` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `acancel()`

Library Functions - `arequest(), aread(), awrite(), listio()`

The POSIX asynchronous I/O calls are:

System Calls - `acancel()`

Library Functions – `iosuspend(), aread(), awrite(), listio(), afsync().`

Example - `iosuspend.c`

NAME

`isatty`, `ttyname`,
`ttyname_r` – determine the nature of a file descriptor

SYNOPSIS

```
int isatty(fd)
int fd;

char *ttyname(fd)
int fd;

int ttyname_r(fd, buf, len)
int fd;
char *buf;
int len;
```

DESCRIPTION

`isatty` uses an `ioctl` call to determine whether the given open file descriptor is attached to a `tty` device. If so `isatty` returns a non-zero result, otherwise zero.

`ttyname` looks through the `/dev` directory in an attempt to match the device number of the file descriptor with a character special file. If a match is found, a pointer to the path name to that special file is returned. If no match is found, or an error is encountered, `ttyname` returns NULL. `ttyname_r` is a reentrant version of `ttyname`; it is functionally identical, but stores its result in **`buf`**, which is of length **`len`**, rather than using a static area that multiple threads can mutilate by concurrent access.

These routines operate on file descriptors, not stream pointers.

NOTE

The path name from `ttyname` is stored in a static array, and so should be copied if it is to be saved.

`ttyname` is slow.

SEE ALSO

System Calls - `ioctl(), stat()`

NAME

isnan, isnand,
ircsnanf – test for floating-point NaN (Not-A-Number)

SYNOPSIS

```
#include <ieeefp.h>

int isnan(dsrc)
double dsrc;

int isnand(dsrc)
double dsrc;

int isnanf(fsrc)
float fsrc;
```

DESCRIPTION

isnan, isnand, and isnanf check if the argument **dsrc** or **fsrc** is a floating-point NaN (Not-a-Number). They return 1 if the argument is a NaN, or 0 if the argument is not a NaN.

isnand is simply a synonym for isnan.

isnanf is a macro defined in ieeefp.h.

NAME

`l3tol, ltol3` – convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol(lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3(cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

`l3tol` converts a list of **`n`** three-byte integers stored in the character array pointed to by **`cp`** into a list of long integers pointed to by **`lp`**.

`ltol3` performs the reverse conversion from a list of long integers pointed to by **`lp`** to a list of three-byte integers stored in the character array pointed to by **`cp`**.

NOTE

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

libexpect – C functions to control programs that are interactive

SYNOPSIS

#include expect_tcl.h
Exp_Init(*interp***);**

cc *files***... -lexpect -ltcl -lm**

DESCRIPTION

This library contains functions that allow Expect to be used as a Tcl extension or to be used directly from C or C++ (without Tcl). Adding Expect as a Tcl extension is very short and simple, so that will be covered first.

The `Exp_Init()` function adds expect commands to the named interpreter. It avoids overwriting commands that already exist, however aliases beginning with`exp_` are always created for expect commands. So for example, `send` can be used as `exp_send`.

Generally, you should only call Expect commands via `Tcl_Eval`. Certain auxiliary functions may be called directly. They are summarized below. They may be useful in constructing your own main. Look at the file `exp_main_exp.c` in the Expect distribution as a prototype main. Another prototype is `tclAppInit.c` in the Tcl source distribution. A prototype for working with Tk is in `exp_main_tk.c` in the Expect distribution.

```
int exp_cmdlinecmds;
int exp_interactive;
FILE *exp_cmdfile;
char *exp_cmdfilename;
int exp_tcl_debugger_available;
void exp_parse_argv(Tcl_Interp *,int argc,char **argv);
int exp_interpreter(Tcl_Interp *);
void exp_interpret_cmdfile(Tcl_Interp *,FILE *);
void exp_interpret_cmdfilename(Tcl_Interp *,char *);
void exp_interpret_rcfiles(Tcl_Interp *,int my_rc,int sys_rc);
char * exp_cook(char *s,int *len);
void (*exp_app_exit)EXP_PROTO((Tcl_Interp *);
```

```
void exp_exit(Tcl_Interp *,int status);
void exp_exit_handlers(Tcl_Interp *);
void exp_error(Tcl_Interp,char *,...);
```

`exp_cmdlinecmds`
is 1 if Expect has been invoked with commands on the program command-line (using **-c** for example).

`exp_interactive`
is 1 if Expect has been invoked with the **-i** flag or if no commands or script is being invoked.

`exp_cmdfile` is a stream from which Expect will read commands.

`exp_cmdfilename`
is the name of a file which Expect will open and read commands from.

`exp_tcl_debugger_available`
is 1 if the debugger has been armed.

`exp_parse_argv` reads the representation of the command line. Based on what is found, any of the other variables listed here are initialized appropriately.

`exp_interpreter`
interactively prompts the user for commands and evaluates them.

`exp_interpret_cmdfile`
reads the given stream and evaluates any commands found.

`exp_interpret_cmdfilename`
opens the named file and evaluates any commands found.

`exp_interpret_rcfiles`
reads and evaluates the `.rc` files. If `my_rc` is zero, then `~/.expectrc` is skipped. If `sys_rc` is zero, then the system-wide expectrc file is skipped.

`exp_cook` returns a static buffer containing the argument reproduced with newlines replaced by carriage-return linefeed sequences. The primary purpose of this is to allow messages to be produced without worrying about whether the terminal is in raw mode or cooked mode. If length is zero, it is computed via strlen.

`exp_error` is a printf-like function that writes the result to `interp->result`.

SYNOPSIS

```
#include <expect.h>

int exp_spawnl(file, arg0 [, arg1, ..., argn] (char *)0);
char *file;
char *arg0, *arg1, ... *argn;

int exp_spawnv(file, argv);
char *file, *argv[ ];

int exp_spawnfd(fd);
int fd;

FILE * exp_popen(command);
char *command;

extern int exp_pid;
extern int exp_ttyinit;

extern int exp_ttycopy;
extern int exp_console;
extern char *exp_stty_init;
extern void (*exp_close_in_child)();
extern void (*exp_child_exec_prelude)();
extern void exp_close_tcl_files();

cc files... -lexpect -ltcl -lm
```

DESCRIPTION

`exp_spawnl` and `exp_spawnv` fork a new process so that its `stdin`, `stdout`, and `stderr` can be written and read by the current process. *file* is the name of a file to be executed. The *arg* pointers are null-terminated strings. Following the style of `execve()`, `arg0` (or `argv[0]`) is customarily a duplicate of the name of the file.

Four interfaces are available, exp_spawnl is useful when the number of arguments is known at compile time. exp_spawnv is useful when the number of arguments is not known at compile time. exp_spawnfd is useful when an open file descriptor is already available as a source. exp_popen is explained later on.

If the process is successfully created, a file descriptor is returned which corresponds to the process's stdin, stdout and stderr. A stream may be associated with the file descriptor by using fdopen(). (This should almost certainly be followed by setbuf() to unbuffer the I/O.)

Closing the file descriptor will typically be detected by the process as an EOF. Once such a process exits, it should be waited upon (via wait) in order to free up the kernel process slot. (Some systems allow you to avoid this if you ignore the SIGCHLD signal).

exp_popen is yet another interface, styled after popen(). It takes a Bourne shell command line, and returns a stream that corresponds to the process's stdin, stdout and stderr. The actual implementation of exp_popen below demonstrates exp_spawnl.

```
FILE *
exp_popen(program)
char *program;
{
  FILE *fp;
  int ec;
  if (0 > (ec = exp_spawnl("sh","sh","-c",program,(char *)0)))
    return(0);
  if (NULL == (fp = fdopen(ec,"r+")) return(0);
  setbuf(fp,(char *)0);
  return(fp);
}
```

After a process is started, the variable exp_pid is set to the process-id of the new process. The variable exp_pty_slave_name is set to the name of the slave side of the pty.

The spawn functions uses a pty to communicate with the process. By default, the pty is initialized the same way as the user's tty (if possible, i.e., if the environment has a controlling terminal.) This initialization can be skipped by setting exp_ttycopy to 0.

The pty is further initialized to some system wide defaults if `exp_ttyinit` is non-zero. The default is generally comparable to `stty sane` .

The tty setting can be further modified by setting the variable `exp_stty_init`. This variable is interpreted in the style of stty arguments. For example, `exp_stty_init = "sane";` repeats the default initialization.

On some systems, it is possible to redirect console output to ptys. If this is supported, you can force the next spawn to obtain the console output by setting the variable `exp_console` to 1.

Between the time a process is started and the new program is given control, the spawn functions can clean up the environment by closing file descriptors. By default, the only file descriptors closed are ones internal to Expect and any marked `close-on-exec`.

If needed, you can close additional file descriptors by creating an appropriate function and assigning it to `exp_close_in_child`. The function will be called after the fork and before the exec. (This also modifies the behavior of the spawn command in Expect.)

If you are also using Tcl, it may be convenient to use the function `exp_close_tcl_files` which closes all files between the default standard file descriptors and the highest descriptor known to Tcl. (Expect does this.)

The function `exp_child_exec_prelude` is the last function called prior to the actual exec in the child. You can redefine this for effects such as manipulating the uid or the signals.

How to allocate your own pty

```
extern int exp_autoallocpty;
extern int exp_pty[2];
```

The spawn functions use a pty to communicate with the process. By default, a pty is automatically allocated each time a process is spawned. If you want to allocate ptys yourself, before calling one of the spawn functions, set `exp_autoallocpty` to 0, `exp_pty[0]` to the master pty file descriptor and `exp_pty[1]` to the slave pty file descriptor. The expect library will not do any pty initializations (e.g., `exp_stty_init` will not be used). The slave pty file descriptor will be automatically closed when the process is spawned. After the process is started, all further communication takes place with the master pty file descriptor.

`exp_spawnl` and `exp_spawnv` duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

Expect processing

While it is possible to use `read()` to read information from a process spawned by `exp_spawnl` or `exp_spawnv`, more convenient functions are provided. They are as follows:

```
int
exp_expectl(fd,type1,pattern1,[re1,],value1,type2,...,exp_end);
int fd;
enum exp_type type;
char *pattern1, *pattern2, ...;
regexp *re1, *re2, ...;
int value1, value2, ...;

int
exp_fexpectl(fp,type1,pattern1,[re1,]value1,type2,...,exp_end);
FILE *fp;
enum exp_type type;
char *pattern1, *pattern2, ...;
regexp *re1, *re2, ...;
int value1, value2, ...;


enum exp_type {
exp_end,
exp_glob,
exp_exact,
exp_regexp,
exp_compiled,
exp_null,
};

struct exp_case {
char *pattern;
regexp *re;
```

```
enum exp_type type;
int value;
};

int exp_expectv(fd,cases);
int fd;
struct exp_case *cases;

int exp_fexpectv(fp,cases);
FILE *fp;
struct exp_case *cases;

extern int exp_timeout;
extern char *exp_match;
extern char *exp_match_end;
extern char *exp_buffer;
extern char *exp_buffer_end;
extern int exp_match_max;
extern int exp_full_buffer;
extern int exp_remove_nulls;
```

The functions wait until the output from a process matches one of the patterns, a specified time period has passed, or an EOF is seen.

The first argument to each function is either a file descriptor or a stream. Successive sets of arguments describe patterns and associated integer values to return when the pattern matches.

The type argument is one of four values. `exp_end` indicates that no more patterns appear. `exp_glob` indicates that the pattern is a glob-style string pattern. `exp_exact` indicates that the pattern is an exact string. `exp_regexp` indicates that the pattern is a regexp-style string pattern. `exp_compiled` indicates that the pattern is a regexp-style string pattern, and that its compiled form is also provided. `exp_null` indicates that the pattern is a null (for debugging purposes, a string pattern must also follow).

If the compiled form is not provided with the functions `exp_expectl` and `exp_fexpectl`, any pattern compilation done internally is thrown away after the function returns. The functions `exp_expectv` and `exp_fexpectv` will automatically compile patterns and will not throw them away. Instead, they must be discarded by the user, by calling free on each pattern. It is only necessary to discard them, the last time the cases are used.

Regexp subpatterns matched are stored in the compiled regexp. Assuming `re` contains a compiled regexp, the matched string can be found in `re->startp[0]`. The match substrings (according to the parentheses) in the original pattern can be found in `re->startp[1]`, `re->startp[2]`, and so on, up to `re->startp[9]`. The corresponding strings ends are `re->endp[x]` where `x` is that same index as for the string start.

The type `exp_null` matches if a null appears in the input. The variable `exp_remove_nulls` must be set to 0 to prevent nulls from being automatically stripped. By default, `exp_remove_nulls` is set to 1 and nulls are automatically stripped.

`exp_expectv` and `exp_fexpectv` are useful when the number of patterns is not known in advance. In this case, the sets are provided in an array. The end of the array is denoted by a struct `exp_case` with type `exp_end`. For the rest of this discussion, these functions will be referred to generically as *expect.*

If a pattern matches, then the corresponding integer value is returned. Values need not be unique, however they should be positive to avoid being mistaken for `EXP_EOF`, `EXP_TIMEOUT`, or `EXP_FULLBUFFER`. Upon EOF or timeout, the value `EXP_EOF` or `EXP_TIMEOUT` is returned. The default timeout period is 10 seconds but may be changed by setting the variable `exp_timeout`. A value of −1 disables a timeout from occurring. A value of 0 causes the expect function to return immediately (i.e., `poll`) after one `read()`. However it must be preceded by a function such as `select`, `poll`, or an event manager callback to guarantee that there is data to be read.

If the variable `exp_full_buffer` is 1, then `EXP_FULLBUFFER` is returned if `exp_buffer` fills with no pattern having matched.

When the expect function returns, `exp_buffer` points to the buffer of characters that was being considered for matching. `exp_buffer_end` points to one past the last character in `exp_buffer`. If a match occurred, `exp_match` points into `exp_buffer` where the match began. `exp_match_end` points to one character past where the match ended.

Each time new input arrives, it is compared to each pattern in the order they are listed. Thus, you may test for absence of a match by making the last pattern something guaranteed to appear, such as a prompt. In situations where there is no prompt, you must check for `EXP_TIMEOUT` (just like you would if you were interacting manually). More philosophy and strategies on specifying `expect` patterns can be found in the documentation on the `expect` program itself. See SEE ALSO below.

Patterns are the usual C-shell-style regular expressions. For example, the following fragment looks for a successful login, such as from a telnet dialogue.

```
switch (exp_expectl(
exp_glob,"connected",CONN,
exp_glob,"busy",BUSY,
exp_glob,"failed",ABORT,
exp_glob,"invalid password",ABORT,
exp_end)) {
case CONN:/* logged in successfully */
break;
case BUSY:/* couldn't log in at the moment */
break;
case EXP_TIMEOUT:
case ABORT:/* can't log in at any moment! */
break;
default: /* problem with expect */
}
```

Asterisks (as in the example above) are a useful shorthand for omitting line-termination characters and other detail. Patterns must match the entire output of the current process (since the previous read on the descriptor or stream). More than 2000 bytes of output can force earlier bytes to be *forgotten*. This may be changed by setting the variable `exp_match_max`. Note that excessively large values can slow down the pattern matcher.

Running in the background

```
extern int exp_disconnected;
int exp_disconnect();
```

It is possible to move a process into the background after it has begun running. A typical use for this is to read passwords and then go into the background to sleep before using the passwords to do real work.

To move a process into the background, fork, call `exp_disconnect()` in the child process and `exit()` in the parent process. This disassociates your process from the controlling terminal. If you wish to move a process into the background in a different way, you must set the variable `exp_disconnected` to 1. This allows processes spawned after this point to be started correctly.

Multiplexing

By default, the expect functions block inside of a read on a single file descriptor. If you want to wait on patterns from multiple file descriptors, use `select`, `poll`, or an event manager. They will tell you what file descriptor is ready to read.

When a file descriptor is ready to read, you can use the expect functions to do one and only read by setting timeout to 0.

Slave control

```
void exp_slave_control(fd,enable)
int fd;
int enable;
```

Pty trapping is normally done automatically by the expect functions. However, if you want to issue an ioctl, for example, directly on the slave device, you should temporary disable trapping.

Pty trapping can be controlled with `exp_slave_control`. The first argument is the file descriptor corresponding to the spawned process. The second argument is a 0 if trapping is to be disabled and 1 if it is to be enabled.

Errors

All functions indicate errors by returning −1 and setting errno.

Errors that occur after the spawn functions fork (e.g., attempting to spawn a non-existent program) are written to the process's stderr, and will be read by the first `expect`.

Signals

```
extern int exp_reading;
extern jmp_buf exp_readenv;
```

`expect` uses `alarm()` to timeout, thus if you generate alarms during `expect`, it will timeout prematurely.

Internally, `expect` calls `read()` which can be interrupted by signals. If you define signal handlers, you can choose to restart or abort `expect`'s internal read. The variable, `exp_reading`, is true if (and only if) an `expect` read function has been interrupted.

```
longjmp(exp_readenv,EXP_ABORT)
```

will abort the read.

```
longjmp(exp_readenv,EXP_RESTART)
```

will restart the read.

Logging

```
extern int exp_loguser;
extern int exp_logfile_all
extern FILE *exp_logfile;
```

If `exp_loguser` is nonzero, `expect` sends any output from the spawned process to stdout. Since interactive programs typically echo their input, this usually suffices to show both sides of the conversation. If `exp_logfile` is also nonzero, this same output is written to the stream defined by `exp_logfile`. If `exp_logfile_all` is non-zero, `exp_logfile` is written regardless of the value of `exp_loguser`.

Debugging

While I consider the library to be easy to use, I think that the standalone expect program is much easier to use than working with the C compiler and its usual edit, compile, debug cycle. Unlike typical C programs, most of the debugging isn't getting the C compiler to accept your programs – rather, it is getting the dialogue correct. Also, translating scripts from expect to C is usually not necessary. For example, the speed of interactive dialogues is virtually never an issue. So please try the standalone `expect` program first. I suspect it is a more appropriate solution for most people than the library.

Nonetheless, if you feel compelled to debug in C, here are some tools to help you.

```
extern int exp_is_debugging;
extern FILE *exp_debugfile;
```

While expect dialogues seem very intuitive, trying to codify them in a program can reveal many surprises in a program's interface. Therefore a variety of debugging aids are available. They are controlled by the above variables, all 0 by default.

Debugging information internal to `expect` is sent to stderr when `exp_is_debugging` is non-zero. The debugging information includes every character received, and every attempt made to match the current input against the patterns. In addition, non-printable characters are translated to a printable form. For example, a control-C appears as a caret followed by a C. If `exp_logfile` is non-zero, this information is also written to that stream.

If `exp_debugfile` is non-zero, all normal and debugging information is written to that stream, regardless of the value of `exp_is_debugging`.

CAVEATS

The stream versions of the `expect` functions are much slower than the file descriptor versions because there is no way to portably read an unknown number of bytes without the potential of timing out. Thus, characters are read one at a time. You are therefore strongly encouraged to use the file descriptor versions of `expect` (although, automated versions of interactive programs don't usually demand high speed anyway).

You can actually get the best of both worlds, writing with the usual stream functions and reading with the file descriptor versions of `expect` as long as you don't attempt to intermix other stream input functions (e.g., `fgetc()`). To do this, pass `fileno(stream)` as the file descriptor each time. Fortunately, there is little reason to use anything but the `expect` functions when reading from interactive programs.

There is no matching `exp_pclose` to `exp_popen` (unlike `popen()` and `pclose()`). It only takes two functions to close down a connection (`fclose()` followed by waiting on the pid), but it is not uncommon to separate these two actions by large time intervals, so the function seems of little value.

If you are running on a Cray running Unicos (all I know for sure from experience), you must run your compiled program as root or setuid. The problem is that the Cray only allows root processes to open ptys. You should observe as much precautions as possible: If you don't need permissions, `setuid(0)` only immediately before calling one of the spawn functions and immediately set it back afterwards.

Normally, `spawn` takes little time to execute. If you notice `spawn` taking a significant amount of time, it is probably encountering ptys that are wedged. A number of tests are run on ptys to avoid entanglements with errant processes. (These take 10 seconds per wedged pty.) Running `expect` with the **-d** option will show if `expect` is encountering many ptys in odd states. If you cannot kill the processes to which these ptys are attached, your only recourse may be to reboot.

BUGS

Follow the advice (above) about using the `exp_expect` functions (which doesn't need to call `getc()`). If you fix the problem (before I do – please check the latest release) let me know.

SEE ALSO

Utility Programs -	`expect`
System Calls -	`execve()`, `read()`, `write()`
Library Functions -	`alarm()`, `fdopen()`, `execvp()`, `longjmp()`
Device Files -	`pty`

There are several examples C programs in the test directory within the `expect` source distribution which use the expect library.

An alternative to this library is the `expect` program. `expect` interprets scripts written in a high-level language which direct the dialogue. In addition, the user can take control and interact directly when desired. If it is not absolutely necessary to write your own C program, it is much easier to use `expect` to perform the entire interaction. It is described further in the following references:

> "expect: Curing Those Uncontrollable Fits of Interactivity" by Don Libes, *Proceedings of the Summer 1990 USENIX Conference,* Anaheim, California, June 11-15, 1990.
>
> "Using expect to Automate System Administration Tasks" by Don Libes, *Proceedings of the 1990 USENIX Large Installation Systems Administration Conference,* Colorado Springs, Colorado, October 17-19, 1990.

AUTHOR

Don Libes, `libes@nist.gov`, National Institute of Standards and Technology

ACKNOWLEDGEMENTS

Thanks to John Ousterhout (UCBerkeley) for supplying the pattern matcher.

Design and implementation of the `expect` library was paid for by the U.S. government and is therefore in the public domain. However the author and NIST would like credit if this program and documentation or portions of them are used.

NAME

`lio_listio` – list directed I/O

SYNOPSIS

```
#include <aio.h>

int lio_listio(mode, list, nent, sig)
int mode;
struct aiocb *list[];
int nent;
struct sigevent *sig;
```

DESCRIPTION

`lio_listio` is used to submit a list of I/O requests with a single function call.

The **mode** argument has two valid values: `LIO_WAIT` and `LIO_NOWAIT`.

`LIO_WAIT` With this value for the mode argument, the `lio_listio` function blocks until all submitted I/O requests have completed. In this case, the **sig** argument is ignored.

`LIO_NOWAIT` With this value for the mode argument, the `lio_listio` function returns immediately. When all the submitted I/O requests have completed, a signal is delivered according to the **sig** argument.

The **list** argument is an array of pointers to **aiocb** structures. The **aiocb** structure is defined in **<aio.h>** as:

Type	Name	Description
`int`	`aio_fildes`	File descriptor
`off_t`	`aio_offset`	File offset
`volatile void *`	`aio_buf`	Pointer to buffer
`size_t`	`aio_nbytes`	Length of data
`int`	`aio_reqprio`	Async I/O priority

Type	Name	Description
`struct sigevent`	`aio_sigevent`	Signal number and value
`int`	`aio_lio_opcode`	List I/O operation

This structure is described in detail in the `aio_read` manual page.

The `aio_lio_opcode` of each `aiocb` structure specifies the I/O operation to be performed for that structure. The valid values for this argument are `LIO_READ`, `LIO_WRITE`, and `LIO_NOP`.

`LIO_READ` An I/O operation is submitted as if by a call to `aio_read` for this **aiocb** structure.

`LIO_WRITE` An I/O operation is submitted as if by a call to `aio_write` for this **aiocb** structure.

`LIO_NOP` This **aiocb** structure is ignored.

The `aio_fildes` member of each **aiocb** structure specifies the file descriptor on which the I/O operation is to be performed. The `aio_buf` member specifies the address of the buffer to or from which the data is to be transferred. The `aio_nbytes` member specifies the number of bytes of data to be transferred.

The array list may contain `NULL` elements, which are ignored. The I/O requests from the array **list** are submitted in an unspecified order.

The **nent** argument specifies the number of elements in **list**.

The **sig** argument is a pointer to a `sigevent` structure, and is relevant when the value of the mode argument is `LIO_NOWAIT`. This `sigevent` structure defines the signal to be generated when all of the submitted I/O requests have completed. If **sig** is `NULL`, or if the member `sig->sigev_signo` is zero, then no signal delivery occurs. If **sig** is not `NULL`, and the `sig->sigev_signo` member is nonzero, then a signal is delivered. Based on the value of this member, this can be a regular signal or a real-time signal. For signal delivery, a proper signal handler, or a real-time signal handler should be set. Refer to the `aio_read` manual page for details on setting up a real-time signal handler.

DIAGNOSTICS

`lio_listio` returns `0` on success. If the mode argument is `LIO_NOWAIT`, it returns when the requests have been successfully queued. If the mode argument is `LIO_WAIT`, it returns when all I/O has been successfully completed. On failure, `lio_listio` returns `-1` and sets **errno** as described below.

The return value of `lio_listio` only indicates the success or failure of `lio_listio` itself. It does not reflect the status of the individual I/O requests. Even if `lio_listio` returns `0` indicating success, one or more of the I/O requests may fail. Success or failure of an individual I/O request is independent of other I/O requests. The return status and error status of each individual I/O operation can be determined with the functions `aio_return` and `aio_error` respectively for the individual **aiocb** structure.

The **errno** values set on failure of `lio_listio` are as follows:

`EAGAIN`	Sufficient resources to queue all I/O operations do not exist. Some I/O operations may have been successfully submitted, and may be determined by the `aio_error` function.
	The number of entries indicated by **nent** would exceed the system-wide limit `AIO_MAX`.
`EINVAL`	The **mode** argument is invalid.
	The value of **nent** is greater than `AIO_LISTIO_MAX`.
`EINTR`	A signal was received while waiting for all I/O requests to finish during a `LIO_WAIT` operation. This signal may be caused by the completion of one or more of the very I/O operations being awaited. Receiving a signal does not necessarily cancel an I/O operation submitted by `lio_listio`, if it is already in progress.
`EIO`	One or more of the individual I/O operations failed.

If `lio_listio` fails with the error `EAGAIN`, `EINTR`, or `EIO`, then it is possible that some of the I/O requests were successfully initiated. The status of each individual I/O request can be checked with the functions `aio_return` and `aio_error` for the individual **`aiocb`** structures. The error codes which can be set are the same as those that can be set by `read` or `write`, with the following additional codes:

EAGAIN	The requested I/O operation was not queued due to lack of resources.
ECANCELED	The requested I/O operation was cancelled by an `aio_cancel` call.
EINPROGRESS	The requested I/O is in progress.

CONFORMANCE

`lio_listio` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls -	`read(), write(), lseek(), fsync()`
Library Calls -	`aio_read(), aio_write(), aio_error(), aio_return()`

NAME

`listio` – perform multiple asynchronous reads and writes

SYNOPSIS

```
#include <sys/aio.h>

listio(mode, liocbps, nent, event)
int mode;
struct liocb *liocbps[];
int nent;
struct event *event;
```

DESCRIPTION

`listio` allows the calling process to initiate multiple asynchronous I/O requests with a single system call.

In some cases an event can be sent for each individual I/O request, plus a "final" event on completion of all I/O requests. The **`event`** argument is used to describe the final event.

The mode argument must be one of: `LIO_WAIT`, `LIO_NOWAIT`, or `LIO_ASYNC`.

`LIO_WAIT` The `listio` call will block until all the I/O initiated has completed; in this case event is ignored. No individual events will be sent for completed I/O.

`LIO_NOWAIT` The call behaves just as if **`mode`** were `LIO_ASYNC`, but no final event is sent; **`event`** is ignored.

`LIO_ASYNC` The `listio` call will return after the I/O has been initiated; when all the I/O has completed, the final event described by **`event`** will be sent to the calling thread.

`liocbps` is an array of pointers to **`liocb`** structures.

`nent` specifies the number of entries (elements) in the array of **`liocb`** structures. Entries define individual I/O operations to be performed.

The **liocb** structure, defined in <sys/aio.h>, contains at least the following fields:

```
int             lio_opcode;
int             lio_fildes;
char            *lio_buf;
unsigned        lio_nbytes;
struct aiocb    lio_aiocb;
```

The `lio_opcode` defines the operation and may have one of three values:

LIO_NOP	The **liocb** is ignored.
LIO_READ	An asynchronous read operation is initiated as if `aread` had been called: `aread (lio_fildes, lio_buff, lio_nbytes, &lio_aiocb);`
LIO_WRITE	An asynchronous write operation is initiated as if `awrite` had been called: `awrite(lio_fildes, lio_buff, lio_nbytes, &lio_aiocb);`

The `lio_fildes` field is the file descriptor for the operation.

The `lio_nbytes` field is the number of bytes to be written or read.

The `lio_buf` field specifies the buffer the data is to be written to or read from.

The `lio_aiocb` field is used as described in the `aread` or `awrite` manual pages (refer to them for the description of the `aiocb` structure). As described in these manual pages, if the `listio` operation is not `LIO_WAIT`, then `aio_errno` in each `aiocb` structure will be `EINPROG` until the operation completes.

The events structure passed on the command line is only used if the `LIO_ASYNC` flag is set in mode. This structure defines the event to be sent on completion of ALL of the `listio` commands. For a description of the events structure and how to use it, see the `aread` or `awrite` manual pages. Note that it is important to use `sigaction` to set the action to be performed on receiving the event.

Note that the individual asynchronous I/O operations still may have their own events to be raised upon individual I/O completion. When **mode** is `LIO_ASYNC`, it is possible to get an event for each individual I/O, plus one when all the individual I/Os have completed.

The I/O submitted by `listio` can be polled using `iosuspend`, `evtpoll`, or `sigwait`.

NOTE

While multiple asynchronous operations are being performed against a file, it is not, in general, possible to accurately determine the file pointer position. In general, all I/O performed on such a (block) file should be performed using `aread`, `awrite`, or `listio`.

Alternatively, (for `awrite` only) the file may be set to append-only mode using `fcntl` to set the `O_APPEND` flag. This has the effect of pegging the seek pointer at the end of the file. Character special files are a different matter since seeks are not allowed on them.

To use this function, include the library `libc_p.a` in the compilation command line.

DIAGNOSTICS

If `listio` succeeds, 0 is returned; if it fails, -1 is returned.

The error returns of this call can be somewhat complicated. The return value of the `listio` call reports only the success or failure of the `listio` call. Individual requests may fail asynchronously, so that `listio` returns successfully, while a check of the `aio_nobytes` fields in the individual **liocb** structures might show that an individual operation failed asynchronously. Thus, to check the success of the call, check the immediate return code and all the asynchronous return codes.

If the mode is `LIO_WAIT`, the call is not handled asynchronously. Failure of an individual operation (for instance, bad `lio_opcode`) would result in the error code `EFAIL`.

Failure of any one of the individual I/O requests will not affect the success/failure of any other I/O request.

The error codes below are used if the `listio` call fails immediately (not asynchronously). If the `listio` call fails, **errno** will contain one of the following:

EAGAIN — The system did not have enough resources to queue all of the `listio` request: all the asynchronous I/O threads are currently busy; no more memory exists for creating new events; no more asynchronous I/O buffers exit for queueing the operation.

EINVAL — **mode** is not one of the three legal values. **nent** is less than or equal to 0.

EFAULT — **liocbps** is a bad address. One of the pointers in **liocbps** is a bad address. `lio_buf` is a bad address.

EFAIL — This **errno** is only set if the **mode** was `LIO_WAIT`. `EFAIL` is used to indicate that an individual operation failed. If the `EFAIL` error code is returned, check the individual `aio_errno` field to determine the exact error. Example: the individual operations can fail because of an illegal `lio_opcode`.

EINTR — The **mode** was `LIO_WAIT` and a signal or event was delivered while waiting for all of the I/O requests to complete. Any outstanding I/O operations are NOT canceled, and the application will need to look at each **liocbps** element to determine the individual I/O operation's status. Since each I/O operation may send its own event, it may be possible to use the event notification to determine which operation completed (if any).

Note that only the first operation which completes and sends an event will cause the `EINTR` error code.

In addition to any errors returned by the `listio` function, the individual I/O operations have their associated error returns as described in the `aread` and `awrite` manual pages. Also, as described in the aread and `awrite` manual pages, the `aio_errno` field is set to `EINPROG` when the request is first submitted; this value may be polled to determine when the asynchronous operation completes. The `aio_errno` is set to either 0 or the appropriate error code when the call completes. Once the call completes, the `aiocb` and any data buffers used in the call can be reused.

CONFORMANCE

`listio` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `lseek(), acancel(), read(), write(), sigwait(), sigaction(), evtpoll(), fcntl()`

Library Functions - `iosuspend(), awrite(), aread()`

The POSIX asynchronous I/O calls are:

System Calls - `acancel()`

Library Functions - `iosuspend(), aread(), awrite(), listio(), afsync().`

Example - `listio.c`

NAME

`localeconv` – determine current locale format rules for numerics

SYNOPSIS

```
#include <locale.h>
struct lconv *localeconv()
```

DESCRIPTION

The `localeconv` function returns a pointer to an lconv structure containing information on the rules used by the current locale to format numeric quantities (numeric and otherwise). The **lconv** structure is defined in `locale.h` as follows:

```
struct lconv {
  char *decimal_point;
  char *thousands_sep;
  char *grouping;
  char *int_curr_symbol;
  char *currency_symbol;
  char *mon_decimal_point;
  char *mon_thousands_sep;
  char *mon_grouping;
  char *positive_sign;
  char *negative_sign;
  char int_frac_digits;
  char frac_digits;
  char p_cs_precedes;
  char p_sep_by_space;
  char n_cs_precedes;
  char n_sep_by_space;
  char p_sign_posn;
  char n_sign_posn;
};
```

The fields with type `char *` are pointers to strings, and all except `decimal_point` can point to “ ” to indicate that the value is not available in the current locale or is of zero length. The fields with type `char` are nonnegative numbers, any of which can be `CHAR_MAX` to indicate that the value is not available in the current locale. The fields have the following meanings:

`char *decimal_point`
: The decimal-point character used to format nonmonetary quantities.

`char *thousands_sep`
: The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

`char *grouping`
: A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

`char *int_curr_symbol`
: The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217:1987. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

`char *currency_symbol`
: The local currency symbol applicable to the current locale.

`char *mon_decimal_point`
: The decimal-point used to format monetary quantities.

`char *mon_thousands_sep`
: The separator for groups of digits before the decimal-point in formatted monetary quantities.

`char *mon_grouping`
: A string whose elements indicate the size of each group of digits in formatted monetary quantities.

`char *positive_sign`
: The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`
: The string used to indicate a negative-valued formatted monetary quantity.

`char int_frac_digits`
: The number of fractional digits (those after the decimal-point) to be displayed in a internationally formatted monetary quantity.

`char frac_digits`
: The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

`char p_cs_precedes`
: Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

`char p_sep_by_spac`
: Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

`char n_cs_precedes`
: Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative formatted monetary quantity.

`char n_sep_by_space`
: Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

`char p_sign_posn`
: Set to a value indicating the positioning of the `positive_sign` for a nonnegative formatted monetary quantity.

`char n_sign_posn`
: Set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity.

The elements of grouping and `mon_grouping` are interpreted according to the following:

`CHAR_MAX` No further grouping is to be performed.

`0`	The previous element is to be repeatedly used for the remainder of the digits.
`other`	The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The values of `p_sign_posn` and `n_sign_posn` are interpreted according to the following:

0	Parentheses surround the quantity and `currency_symbol`.
1	The sign string precedes the quantity and `currency_symbol`.
2	The sign string succeeds the quantity and `currency_symbol`.
3	The sign string immediately precedes the `currency_symbol`.
4	The sign string immediately succeeds the `currency_symbol`.

NOTE

The lconv structure pointed to by the return value of `localeconv` should not be modified by the calling program. However, calls to the `setlocale` function with category `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

SEE ALSO

Library Functions - `setlocale()`

NAME

`logname` – get login name of the user login

SYNOPSIS

```
char *logname()
```

DESCRIPTION

`logname` searches for the `LOGNAME` environment variable and returns a pointer to a null-terminated string containing its value.

NOTE

The pointer returned by logname points to static data whose content is overwritten by each call.

SEE ALSO

Library Functions - `getenv()`

NAME

lsearch, lfind – linear search a table and update

SYNOPSIS

```
char *lsearch(key, base, nelp, key_size, compar)
char *key;
char *base;
unsigned *nelp;
int key_size;
int (*compar)();
```

```
char *lfind(key, base, nelp, key_size, compar)
char *key;
char *base;
unsigned *nelp;
int key_size;
int (*compar)();
```

DESCRIPTION

lsearch performs a linear search on a table. **key** points to a data item that is being searched for. **base** points to the element at the base of the table. **nelp** points to an integer containing the current number of elements in the table. **key_size** specifies the size of each element in the table. **compar** points to a function which will be called to compare elements during the search. **compar** is invoked as follows:

```
compar(item1, item2)
itemtype *item1, *item2;
```

compar must return a number less than, equal to, or greater than zero, reflecting the relationship between *item1 and *item2. lsearch returns a pointer into a table indicating where a data may be found. If the data does not occur, it is added at the end of the table, and the integer pointed to by **nelp** is incremented.

lfind performs the same function as lsearch except that if the data is not found, it is not added to the table, and a NULL pointer is returned.

DIAGNOSTICS

If a match is found, both `lsearch` and `lfind` return a pointer to the matched item. Otherwise, `lsearch` returns a pointer to the added item, and `lfind` returns NULL.

NOTE

Severely bad things happen if there is not enough room in the table to add a new item.

SEE ALSO

Library Functions - `bsearch()`, `hsearch()`, `tsearch()`, `qsort()`

NAME

`malloc`, `free`,
`calloc`, `realloc` – memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size)
unsigned long size;

void free (ptr)
void *ptr;

void *calloc(nelem, elsize)
unsigned long nelem, elsize;

void *realloc(ptr, size)
void *ptr;
unsigned long size;
```

DESCRIPTION

`malloc` returns a pointer to a block of at least **size** bytes aligned for any use.

`free` makes space pointed to by **ptr** (formerly allocated by `malloc` or `calloc`) available for further allocation.

`calloc` allocates space for an array of **nelem** elements of size **elsize**. The space is initialized to zeros.

`realloc` adjusts the amount of space allocated at location **ptr** to **size** bytes. The contents of the space may be moved, but are guaranteed to remain unchanged up to the minimum of the old and new sizes. The address of the adjusted space is returned.

NOTE

The contents of the space are no longer valid after `free` returns. Programs which attempt to use the space after it has been freed may give invalid results or core dump. This is also true for space which has been freed by `realloc` when it is called to decrease the size of the space pointed to by **`ptr`**.

DIAGNOSTICS

`malloc`, `realloc`, `calloc`, return `NULL` if no memory is available.

SEE ALSO

System Calls - `brk(), sbrk(), nsbrk()`

NAME

`mbchar: mbtowc,`
`mblen,`
`wctomb –` multibyte character handling

SYNOPSIS

```
#include <stdlib.h>

int mbtowc(pwc, s, n)
wchar_t *pwc;
const char *s;
size_t n;

int mblen(s, n)
const char *s;
size_t n;

int wctomb(s, wchar)
char *s;
wchar_t wchar;
```

DESCRIPTION

Multibyte characters are used in defining extended character sets for locales which have more characters than can be represented using 8 bits.

The multibyte character handling functions are used to translate multibyte characters into wide characters and back again. Wide characters have type **wchar_t** (defined in `<stdlib.h>`), which is an integral type large enough to distinctly encode every member of the largest supported extended character set.

The number of bytes in an extended character set is defined by the `LC_CTYPE` category of the locale. The maximum number of bytes in any multibyte character is `MB_LEN_MAX`, defined in `<limits.h>`. The maximum number of bytes in a multibyte character in an extended character set in the current locale is `MB_CUR_MAX`, defined in `<stdlib.h>`.

If s is not a NULL pointer, mbtowc determines the number of bytes that comprise the multibyte character pointed to by **s**. If **pwc** is not a NULL pointer, mbtowc also converts the multibyte character to a wide character and stores it in the object pointed to by **pwc**. The value of the code corresponding to the NULL character is zero. At most **n** bytes of the array pointed to by **s** will be examined.

If **s** is a NULL pointer, or if **s** points to the NULL character, mbtowc returns 0. If the first **n** or fewer bytes pointed to by **s** form a valid multibyte character, mbtowc returns the number of bytes that comprise the converted multibyte character. Otherwise, **s** does not point to a valid multibyte character, and mbtowc returns -1.

The mblen function determines the number of bytes that comprise the multibyte character pointed to by **s**. It is equivalent to

```
mbtowc((wchar_t *)0, s, n);
```

The wctomb function determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is **wchar**. If **s** is not a NULL pointer, wctomb also stores the multibyte character representation in the array pointed to by **s**. At most MB_CUR_MAX characters are stored.

If **s** is a NULL pointer, wctomb returns 0. Otherwise, wctomb returns -1 if the value of **wchar** does not correspond to a valid multibyte character, or returns the number of bytes that comprise the multibyte character corresponding to the value of **wchar**.

SEE ALSO

Library Functions - localeconv(), mbstring, setlocale()

NAME

mbstring: mbstowcs,
wcstombs – multibyte string handling

SYNOPSIS

```
#include <stdlib.h>

size_t mbstowcs(pwcs, s, n)
wchar_t *pwcs;
const char *s;
size_t n;

size_t wcstombs(s, pwcs, n)
char *s;
const wchar_t *pwcs;
size_t n;
```

DESCRIPTION

The `mbstowcs` function converts a sequence of multibyte characters from the array pointed to by **s** into a sequence of corresponding codes and stores not more than **n** codes into the array pointed to by **pwcs**. No multibyte characters that follow a NULL character (which is converted into a code with value zero) will be examined or converted. If an invalid multibyte character is encountered, `mbstowcs` returns (**size_t**)(-1). Otherwise, `mbstowcs` returns the number of array elements modified, not including the terminating zero code, if any.

The `wcstombs` function converts a sequence of codes that correspond to multibyte characters from the array pointed to by **pwcs** into a sequence of multibyte characters and stores these multibyte characters into the array pointed to by **s**, stopping if a multibyte character would exceed the limit of **n** total bytes or if a NULL character is stored. If a code is encountered that does not correspond to a valid multibyte character, `wcstombs` returns (**size_t**)(-1). Otherwise, `wcstombs` returns the number of bytes modified, not including a terminating NULL character, if any.

SEE ALSO

Library Functions - `localeconv(), mbchar, setlocale()`

NAME

`memory: memccpy,`
`memchr, memcmp,`
`memcpy, memmove, memset` – memory operations

SYNOPSIS

```
#include <string.h>

void *memccpy(s1, s2, c, n)
void *s1;
const void *s2;
int c;
size_t n;

void *memchr(s, c, n)
const void *s;
int c;
size_t n;

int memcmp(s1, s2, n)
const void *s1;
const void *s2;
size_t n;

void *memcpy(s1, s2, n)
void *s1,
const void *s2;
size_t n;

void *memmove(s1, s2, n)
void *s1,
const void *s2;
size_t n;

void *memset(s, c, n)
void *s;
int c;
size_t n;
```

DESCRIPTION

The arguments **s1**, **s2**, and **s** point to arrays of bytes in memory (not necessarily strings, which are always null-terminated). The functions `memccpy` and `memcpy` alter the memory pointed to by **s1**, and the function `memset` alters the memory pointed to by **s**. None of these functions checks for overflow.

`memccpy`	Copies from memory area **s2** to **s1**, up to and including the first occurrence of character **c**, or up to **n** characters, whichever comes first. If **c** is encountered, `memccpy` returns a pointer to the character following **c** in **s1**. Otherwise, it returns a null pointer.
`memchr`	Returns a pointer to the first occurrence of character **c** in memory area **s**, as long as it occurs within the first **n** characters. Otherwise, it returns a null pointer.
`memcmp`	Compares its first 2 arguments and returns an integer less than, equal to, or greater than 0, depending on whether the first **n** characters of **s1** are lexicographically less than, equal to, or greater than the first **n** characters of **s2**.
`memmove`	Copies **n** characters from memory area **s2** to **s1**, which may overlap, and returns **s1**. This function detects overlapping memory areas and copies in the appropriate direction to obtain the expected result.
`memcpy`	Copies **n** characters from memory area **s2** to **s1**, and returns **s1**.
`memset`	Sets the first **n** characters in memory area **s** to the value **c**, and returns **s**.

NOTE

All of the memory functions are declared with appropriate types in **string.h**.

SEE ALSO

Library Functions - `bstring(), string()`

NAME

`mkfifo` – make a FIFO special file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(path mode)
char *path;
mode_t mode;
```

DESCRIPTION

The `mkfifo()` routine creates a new FIFO special file named by the pathname pointed to by **path**. The file permission bits of the new FIFO are initialized from **mode**. The file permission bits of the **mode** argument are modified by the file creation mask of the process (see `umask`). Bits in **mode** other than the file permission bits are ignored.

The owner ID of the FIFO is set to the effective user ID of the process. The group ID of the FIFO is set to the group ID of the directory in which the FIFO is being created.

Upon successful completion, the `mkfifo()` function shall mark for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the file (see `stat`). Also, the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update.

DIAGNOSTICS

Upon successful completion, a value of zero is returned. Otherwise, a value of `-1` is returned, no FIFO is created, and **errno** is set to indicate the error.

If any of the following conditions occur, the `mkfifo()` function shall return `-1` and set **errno** to the corresponding value;

`EACCES`	Search permission is denied for a component of the path prefix or write permission is denied on the parent directory of the file to be created

EEXIST	The named file already exists.
ENAMETOOLONG	The length of the **path** string exceeds PATH_MAX, or a pathname component is longer than {NAME_MAX} while _POSIX_NO_TRUNC is in effect.
ENOENT	A component of the **path** prefix does not exist, or the **path** argument points to an empty string.
ENOSPC	The directory that would contain the new file cannot be extended, or the file system is out of file allocation resources.
ENOTDIR	A component of the **path** prefix is not a directory.
EROFS	The named file resides on a read-only file system.

COPYRIGHT NOTICE

SEE ALSO

System Calls - `chmod()`, `execve()`, `pipe()`, `stat()`, `umask()`

NAME

`mktemp` – create a file name

SYNOPSIS

```
char *mktemp(t)
char *t;
```

DESCRIPTION

`mktemp` creates a file name from a template provided in the string **t**. The template should be a LynxOS pathname, and should include an occurrence of the characters `%` (percent) and `@` (commercial "at" sign). The percent sign is replaced by the current process ID number, and the "at" sign is replaced by the current time stamp as returned by `time`. Filenames created by `mktemp` are ideal for use as temporary file names.

For compatibility with older systems, a string of X characters is treated as a single percent sign.

NOTE

The filename is created in a static buffer which is overwritten by each call.

SEE ALSO

System Calls - `getpid(), open()`

Library Functions - `tmpfile(), tmpnam()`

NAME

`mktimer`, `rmtimer` – create and delete an interval timer

SYNOPSIS

```
#include <sys/timers.h>

#include <sys/events.h>

timer_t mktimer(clock_type, notify_type, itimercbp)
int clock_type;
int notify_type;
struct itimercb *itimercbp;

#include <sys/timers.h>

int rmtimer(timerid)
timer_t timerid;
```

DESCRIPTION

`mktimer` creates an interval timer of the type specified by **clock_type**. `mktimer` returns a unique identifier that is used when setting and querying the timer. `rmtimer` deletes a timer created by `mktimer`.

Interval timers are based on the system wide `TIMEOFDAY` (see `getclock`) clock. Time is continually advanced, whether or not the process that created the timer is running. There is a single timer available to each process. Timers are not shared between processes.

A process may request notification in one of two forms: `DELIVERY_SIGNALS` or `DELIVERY_EVENTS`. The **notify_type** parameter specifies which type of timer to create. The notification type specifies how the task will find out the timer expiration, either through a signal or an event.

If the delivery type is `DELIVERY_SIGNALS`, the system sends a `SIGALRM` signal to the process whenever the timer expires. The requesting process must create a signal handler for the signal; otherwise, the default action will be invoked. The **itimercbp** parameter is ignored.

If the delivery type is `DELIVERY_EVENTS`, the system raises a timer event for the process. **itimercbp** specifies the event to raise and allocates a space for timer overrun count. The structure of the **itimercb** is:

```
struct itimercb {
  struct event itcb_event   /* timer event definition */
  int itcb_count;           /* timer overrun count */
}
```

The event structure is:

```
struct event {
  void (*evt_handler)();  /* event handler */
  void *evt_value;        /* value to pass to event handler*/
  evt_class_t evt_class;  /* event number */
  evtset_t evt_classmask; /*event mask when event is raised*/
}
```

When a timer expires and the event notification is delayed, the number of notifications that were missed are stored in the `itcb_count` field. If no deliveries are missed, this field will be zero. When the event is actually delivered to the process, the field will be updated. The maximum value of the count field is `DELAYTIMERMAX`. If more events than `DELAYTIMERMAX` occur, `itcb_count` will contain `DELAYTIMERMAX`.

`rmtimer` deletes a previously allocated timer. Any pending timer events will be canceled.

DIAGNOSTICS

`mktimer` returns a unique **timerid** if successful. `rmtimer` returns 0 if successful. A return value of -1 indicates that an error occurred, and **errno** is set to one of the following values:

`EAGAIN`	The process has already created an interval timer.
`EINVAL`	**timerid** is not a valid timer id.
`EINVAL`	**clock_type** is not `TIMEOFDAY`.

CONFORMANCE

`mktimer` and `rmtimer` are implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`abstimer()`, `gettimer()`, `reltimer()`, `resabs()`, `resrel()`
Example -	`mktimer.c`, `rmtimer.c`

NAME

`mlock`, `munlock` – lock/unlock a range of process address space

SYNOPSIS

```
#include <sys/mman.h>

int mlock(addr, len)
const void *addr;
size_t len;

int munlock(addr, len)
const void *addr;
size_t len;
```

DESCRIPTION

`mlock` locks a specified range of pages, which are part of the calling process' address space. Pages starting at the address **addr** in the process' address space, and continuing for **len** bytes are locked. Locking these pages makes them memory resident, until the process unlocks them, or until the process exits or execs another process. Locked pages cannot be swapped out even if virtual memory is running. If **addr** is not aligned to a page address, `mlock` and `munlock` truncate it to the nearest page.

`munlock` unlocks a specified range of pages, which are part of the calling process' address space. Pages starting at the address **addr** in the process' address space, and continuing for **len** bytes are unlocked. The **addr** argument should be a multiple of the page size, `PAGESIZE`. The effect is the same whether a page is locked multiple times with `mlock` or only once; it can still be unlocked with a single `munlock` call. If mapped pages are shared between the calling process and some other process, `munlock` does not affect any locks established by the other process.

NOTE

Once a page is locked, it will remain in memory, even if `munlock` is called.

DIAGNOSTICS

`mlock` and `munlock` lock and unlock the specified range of the process address space respectively, when successful and return a value of `0`. On failure these functions return `-1` and set **`errno`** to one of the following:

`EAGAIN`	Some or all of the requested memory could not be locked with the `mlock` call.
`ENOMEM`	Locking memory with `mlock` would exceed the limit on the amount of memory a process may lock.
	Some part of the address range specified by the **`addr`** and **`len`** arguments does not correspond to valid mapped pages in the process' address space.
`EPERM`	The process does not have permission to lock memory with `mlock`.

CONFORMANCE

`mlock` and `munlock` are implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `mlockall(), munlockall()`

NAME

`mlockall`, `munlockall` – lock or unlock address spaces of processes

SYNOPSIS

```
#include <sys/mman.h>

int mlockall(flags)
int flags;

int munlockall(void)
```

DESCRIPTION

The `mlockall` function locks all the pages of the calling process into memory. Locking memory forces the pages to be memory resident. These pages cannot be swapped out even if virtual memory is running. The pages remain memory resident until the process `exit`s or `exec`s another process. The **`flags`** argument indicates whether the pages to be locked are those currently mapped by the process, or those that will be mapped in the future. The **`flags`** argument may be constructed from the bitwise inclusive OR of the following flags:

`MCL_CURRENT`	Lock all of the pages currently mapped into the process' address space.
`MCL_FUTURE`	Lock all of the pages that become mapped into the process' address space in the future, and all the pages currently mapped.

The `munlockall` function unlocks all of the currently mapped pages of the calling process. Unlocking the pages implies that those pages can be swapped out if necessary, when virtual memory is running. If a previous `mlockall` call requested locking of future mapped pages, after the `munlockall` call this request is nullified. If mapped pages are shared between the calling process and some other process, the `munlockall` call does not affect any locks established by the other process.

DIAGNOSTICS

`mlockall` and `munlockall` lock and unlock the process address space respectively, when successful and return a value of `0`. On failure these functions return `-1` and set **`errno`** to one of the following:

`EAGAIN`	Some or all of the memory could not be locked with the `mlockall` call.
`EINVAL`	The **`flags`** argument supplied to the `mlockall` function is zero, or includes invalid flags.
`ENOMEM`	Locking memory with `mlockall` would exceed the limit on amount of memory a process may lock.
`EPERM`	There is no permission to lock memory with `mlockall`.

CONFORMANCE

`mlockall` and `munlockall` are implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `mlock(), munlock()`

NAME

`mq_close` – close a message queue

SYNOPSIS

```
#include <mqueue.h>

int mq_close(mqdes)
mqd_t mqdes;
```

DESCRIPTION

`mq_close` closes the message queue with the message queue descriptor **mqdes**.

If a notification request has been attached with `mq_notify` to **mqdes** by the current process, it is cancelled by `mq_close`.

DIAGNOSTICS

`mq_close` closes the message queue when successful and returns a value of `0`. Otherwise, it returns `-1`, and sets **errno** to one of the following values:

EBADF	The **mqdes** argument is not a valid message queue descriptor.

CONFORMANCE

`mq_close` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `mq_open(), mq_notify()`

NAME

`mq_getattr` – obtain the message queue attributes

SYNOPSIS

```
#include <mqueue.h>

int mq_getattr(mqdes, mqstat)
mqd_t mqdes;
struct mq_attr *mqstat;
```

DESCRIPTION

`mq_getattr` obtains the attributes for the message queue specified by the **mqdes** argument in the location referenced by the **mqstat** argument.

The structure `mq_attr` is defined as follows:

Type	Name	Description
long	mq_flags	Message queue flags
long	mq_maxmsg	Maximum number of messages
long	mq_msgsize	Maximum message size
long	mq_curmsgs	Number of messages currently queued

The meaning of the various members is described below. The following member can be set at creation or set with the `mq_setattr` call at any time. It can be retrieved with the `mq_getattr` call.

`mq_flags` The message queue flags specify the blocking or non-blocking behavior of a message queue. If the `O_NONBLOCK` flag is set, the message queue is non-blocking. Otherwise, it is blocking. A blocking message queue causes the `mq_send` and `mq_receive` functions to block on full and empty message queues, respectively. A non-blocking message queue causes the `mq_send` and `mq_receive` functions to fail on full and empty message queues, respectively.

The following members can only be set at the message queue creation, and can be retrieved with the `mq_getattr` call.

`mq_maxmsg`	This member specifies the maximum number of messages that can be allowed in the message queue. When the message queue contains `mq_maxmsg` messages, it is considered to be full. An `mq_send` call on a blocking message queue, when it is full blocks.
`mq_msgsize`	This member specifies the maximum size of each message in the message queue, in bytes.

The following members provide the status information about a message queue. These members cannot be set, and can only be retrieved with the `mq_getattr` call.

`mq_curmsgs`	This member indicates the number of messages currently on the message queue.

DIAGNOSTICS

`mq_getattr` obtains the attributes for the message queue when successful and returns `0`. On failure, it returns `-1` and sets **`errno`** as follows:

`EBADF`	The **`mqdes`** argument is an invalid message queue descriptor.

CONFORMANCE

`mq_getattr` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `mq_open(), mq_setattr()`

NAME

`mqgetpid` – for a given message and `msgcb`, get a sender PID

SYNOPSIS

```
#include <sys/mqueue.h>

pid_t mqgetpid(msgcbp)
struct msgcb *msgcbp;
```

DESCRIPTION

`mqgetpid` is used to get the process identifier of the process that sent the message corresponding to the specified message queue control block, **msgcb**.

The contents of the **msgcb** structure are described in the `mqsend` manual page.

If the contents of the **msgcb** structure have been modified since its receipt, or if the **msgcb** structure has not been used to receive a message, then the value returned from the `mqgetpid` call will not be valid; an error may or may not be returned. In any case, the result will not be as desired.

DIAGNOSTICS

`mqgetpid` returns the process identifier (**pid**) of the process that sent the message corresponding to **msgcb** when successful and -1 when unsuccessful. If the call fails, **errno** will contain one of the following:

ESRCH	There is no valid process identifier for the process that sent the message containing the specified **msgcb** structure.
EINVAL	The **msgcb** equals zero.

SEE ALSO

System Calls - `mqsend(), mqreceive()`

Calls that operate on POSIX message queue special files include:

System Calls - `mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(), msgalloc(), msgfree(), mqputevt(), mqgetevt()`

Library Functions - `mqgetpid()`

NAME

mq_notify – send a signal upon availability of a message

SYNOPSIS

```
#include <mqueue.h>

int mq_notify(mqdes, notification)
mqd_t mqdes;
const struct sigevent *notification;
```

DESCRIPTION

The mq_notify function is used to attach a notification request by the calling process for the message queue specified by the **mqdes** descriptor. If the **notification** argument is not NULL, then the signal specified by this argument is sent to the calling process when the message queue changes from empty to non-empty. Any message queue can have at most one notification request attached to it. Hence, if some process (including the calling process) has already attached a notification request for this message queue, the mq_notify call fails.

If one or more processes blocked in mq_receive are waiting to receive from the empty message queue, and if the notification request is attached with mq_notify, then the notification request remains pending, and one of the processes blocked in mq_receive is unblocked to receive the message.

The **notification** argument is used to define the asynchronous notification that will occur when the message queue changes from empty to non-empty. The structure sigevent consists of the following members:

Type	Name	Description
int	sigev_signo	Signal number
union sigval	sigev_value	Signal value
int	sigev_notify	Notification type

The calling process may or may not choose to set a POSIX.1b real-time signal handler for the notification request. If such a signal handler is set, a real-time signal is queued to the calling process when the message queue changes from empty to non-empty. Refer to the `sigqueue` manual page for more information on setting up a real-time signal handler.

If the **`notification`** argument is `NULL`, then it has the effect of detaching any previously attached notification request by the calling process for the specified message queue.

DIAGNOSTICS

`mq_notify` sets the notification request for the calling process when successful, and returns `0`. On failure it returns `-1`, and sets **`errno`** as follows:

`EBADF`	The **`mqdes`** argument is an invalid message queue descriptor.
`EBUSY`	Some process has already attached a notification request to this message queue.

CONFORMANCE

`mq_notify` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `mq_send(), mq_recieve()`

NAME

mq_open – open/initialize a message queue

SYNOPSIS

```
#include <mqueue.h>

mqd_t mq_open(name, oflags [,mode, attr])
const char *name;
int oflags;
mode_t mode;
mq_attr *attr;
```

DESCRIPTION

mq_open opens a message queue referenced by the character string **name**, depending on the argument **oflags**. The open message queue with the name **name** can be referenced using the descriptor returned from mq_open. The message queue remains usable by the process that opened it, until it is successfully closed by calls to mq_close, exit, or exec.

The **name** argument must conform to standard pathname construction rules, even though it does not appear in the file system.

The **oflags** argument determines whether the message queue is initialized or merely opened. The **oflags** argument is determined from the bitwise inclusive OR of values from the following list. Exactly one of the first three values can be specified:

O_RDONLY The message queue is opened for receiving messages. The returned message queue descriptor may be used with mq_receive, but not mq_send.

O_WRONLY The message queue is opened for sending messages. The returned message queue descriptor may be used with mq_send, but not mq_receives.

O_RDWR The message queue is opened for both sending and receiving messages.

Any combination of the following values may be used in the **oflags** argument:

O_CREAT — If the message queue does not exist, create it. If the message queue exists, there is no effect except as noted under O_EXCL below. This flag requires two additional arguments: **mode** and **attr**. If the **attr** argument is NULL, the message queue is created with default message queue attributes. If **attr** is non-NULL, the message queue is created with the mq_flags, mq_maxmsg and mq_msgsize attributes of the message queue set to the corresponding members referred to by **attr**.

O_EXCL — If O_EXCL and O_CREAT are both set, mq_open fails if the message queue already exists.

O_NONBLOCK — Setting this flag causes subsequent mq_send and mq_receive calls with the open message queue descriptor to not block when the message is full or empty, respectively.

When creating a message queue, the attributes of the message queue are specified with the following mq_attr structure:

Type	Name	Description
long	mq_flags	Message queue flags
long	mq_maxmsg	Maximum number of messages
long	mq_msgsize	Maximum message size
long	mq_curmsgs	Number of messages currently queued

Only the mq_flags, mq_maxmsg, and mq_msgsize attributes of this structure are relevant when creating the message queue. Setting the mq_flags attribute to O_NONBLOCK has the same effect as creating the message queue with the O_NONBLOCK flag. That is, any mq_send or mq_receive calls to a message queue do not block when it is full or empty respectively.

DIAGNOSTICS

mq_open opens a message queue when successful, and returns 0. Otherwise, it returns (mqd_t)-1, and sets **errno** to one of the following values:

EACCES	The calling process does not have permission to create or open the named message queue.
EEXIST	O_CREAT and O_EXCL are set in **oflags** and the message queue already exists.
EINTR	A signal interrupted the mq_open operation.
EINVAL	Either the mq_maxmsg or mq_msgsize members of the **attr** argument are zero or negative, an O_CREAT is set in **oflags**.
EMFILE	Too many message queue descriptors are currently in use by this process.
ENAMETOOLONG	The **name** argument is too long, or contains pathname elements that are too long
ENFILE	Too many message queues are currently open in the system.
ENOENT	O_CREAT is not set and the named message queue does not exist.
ENOSPC	There is insufficient space for the creation of the new message queue.
ENOSYS	The underlying kernel support required for message queues is not configured. Message queues require POSIX.1b shared memory, POSIX.1b semaphores, and the POSIX.1b name server.

CONFORMANCE

mq_open is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `mq_close(), mq_receive(), mq_send(), mq_notify(), mq_setattr(), mq_getattr(), mq_unlink()`

NAME

`mq_receive` – receive a message from a message queue

SYNOPSIS

```
#include <mqueue.h>

ssize_t mq_receive(mqdes, msg_ptr, msg_len, msg_prio)
mqd_t mqdes;
char *msg_ptr;
size_t msg_len;
unsigned int *msg_prio;
```

DESCRIPTION

`mq_receive` receives a message from the specified message queue. If the message queue specified by the **mqdes** argument is not empty, `mq_receive` removes the oldest of the highest priority message from this message queue. The **msg_ptr** argument specifies the address to which the received message is copied. The **msg_len** argument specifies the size of this buffer (**msg_ptr**) that will hold the message in bytes. It must be larger than the `mq_msgsize` attribute of the specified message queue, or `mq_receive` will fail.

If the `msg_prio` argument is not `NULL`, it is used to hold the priority of the received message.

If the message queue specified by **mqdes** is empty, and if it is non-blocking (it was opened with the `O_NONBLOCK` flag or the `O_NONBLOCK` flag is set in its attributes), `mq_receive` fails and does not receive any message. If the specified message queue is empty, and if it is blocking (it was not opened with the `O_NONBLOCK` flag and the `O_NONBLOCK` flag is not set in its attributes), `mq_receive` blocks until a message becomes available on the message queue. A blocked `mq_receive` call may be interrupted by a signal. If more than one process is blocked to receive a message on an empty message queue, the process with the highest priority which has been waiting the longest is unblocked when a message becomes available.

DIAGNOSTICS

`mq_receive` removes a message from the specified message queue, and returns the length of the message in bytes when successful. On failure, it returns `-1` and sets **errno** as follows:

EAGAIN	The message queue is empty, and the message queue is non-blocking.
EBADF	The **mqdes** argument is not a valid message queue descriptor.
EMSGSIZE	The message buffer size specified by the **msg_len** argument is less than the `mq_msgsize` attribute of the message queue.
EINTR	A signal interrupted the `mq_receive` operation.

CONFORMANCE

`mq_receive` is implemented according to the POSIX 1003.1b (Real-Time) standard.

NOTE

The message priority has no relationship with the process scheduling priority. Message priority determines the order in which messages are added to (and removed from) a message queue. The process priority does however come into picture when more than one processes are blocked on the same full message queue.

SEE ALSO

Library Calls - `mq_send()`, `mq_setattr()`, `mq_selective_receive()`

NAME

mq_selective_receive – receive a message with priority selection

SYNOPSIS

```
#include <mqueue.h>

ssize_t mq_selective_receive(mqdes, msg_ptr, msg_len,
want_prio, msg_prio)
mqd_t mqdes;
char *msg_ptr;
size_t msg_len;
unsigned int want_prio;
unsigned int *msg_prio;
```

DESCRIPTION

mq_selective_receive receives a message of a specified priority or range of priorities from the specified message queue. The **want_prio** argument specifies which priority or range of priorities to receive. If the MQ_MATCH_GE flag is set in **want_prio**, a message whose priority is greater than or equal to the value in **want_prio** will be matched. If MQ_MATCH_GE is not set in **want_prio**, then only messages whose priority is exactly the value in **want_prio** will be matched.

If the message queue specified by the **mqdes** argument contains a message whose priority matches **want_prio**, mq_selective_receive removes the oldest matching message from the message queue. The **msg_ptr** argument specifies the address to which the received message is copied. The **msg_len** argument specifies the size of this buffer (**msg_ptr**) that will hold the message, in bytes. It must be larger than the mq_msgsize attribute of the specified message queue, or mq_selective_receive will fail.

If the **msg_prio** argument is not NULL, the priority of the received message is copied into the location pointed to by **msg_prio**.

If the message queue specified by **mqdes** does not contain any messages that match **want_prio**, and if the queue is non-blocking (it was opened with the O_NONBLOCK flag, or the O_NONBLOCK flag is set in its attributes), mq_selective_receive fails and does not receive any message.

If the specified message queue contains no messages that match **want_prio**, and if the queue is blocking (it was not opened with the O_NONBLOCK flag and the O_NONBLOCK flag is not set in its attributes), mq_selective_receive blocks until a message which matches **want_prio** becomes available on the message queue. A blocked mq_selective_receive call may be interrupted by a signal.

If more than one process is blocked to receive a message on an empty message queue, the process with the highest scheduling priority which has been waiting the longest is unblocked first when a message becomes available. If that process does not receive the message, the next-highest priority process will be awakened, and so forth, until the message gets received, or all processes that are waiting have had a chance to receive the message.

DIAGNOSTICS

mq_selective_receive removes a message from the specified message queue, and returns the length of the message in bytes when successful. On failure, it returns -1 and sets **errno** as follows:

EAGAIN	The message queue contains no matching messages, and the queue is non-blocking.
EBADF	The **mqdes** argument is not a valid message queue descriptor.
EINVAL	The **want_prio** argument specifies a value outside the range of 0..MQ_PRIO_MAX.
EMSGSIZE	The message buffer size specified by the **msg_len** argument is less than the mq_msgsize attribute of the message queue.
EINTR	A signal interrupted the mq_selective_receive operation.

CONFORMANCE

mq_selective_receive is a Lynx proprietary extension to the POSIX 1003.1b (Real-Time) standard.

NOTE

The message priority has no relationship with the process scheduling priority. Message priority determines the order in which messages are added to (and removed from) a message queue. The process priority does, however come into picture when more than one process is blocked on the same full message queue.

SEE ALSO

Library Calls - `mq_receive(), mq_send(), mq_setattr()`

NAME

mq_send – send a message to a message queue

SYNOPSIS

```
#include <mqueue.h>

int mq_send(mqdes, msg_ptr, msg_len, msg_prio)
mqd_t mqdes;
const char *msg_ptr;
size_t msg_len;
unsigned int msg_prio;
```

DESCRIPTION

mq_send sends the message referenced by the **msg_ptr** argument to the message queue specified by the **mqdes** descriptor. The **msg_len** argument specifies the length of the message in bytes. The **msg_prio** argument specifies the priority of the message. The **msg_len** argument must be less than or equal to the **mq_msgsize** attribute of the message queue.

If the message queue is not full, the message is inserted into the message queue in priority order. That is, a message with a larger priority is inserted before messages with lower priorities. Messages of equal priority are inserted in a first-in-first-out order.

If the message queue is full and if it is non-blocking (it was opened with the O_NONBLOCK flag, or the O_NONBLOCK flag is set in its attributes), an error occurs and no message is sent. If the message queue is full and if it is blocking (it was not opened with the O_NONBLOCK flag, and the O_NONBLOCK flag is not set in its attributes), then mq_send blocks until the message queue has space for the message. A signal may interrupt a blocked mq_send call. If more than one process is blocked on the same full message queue, when space becomes available the process with the highest priority which has been waiting the longest is unblocked.

DIAGNOSTICS

mq_send sends the specified message to the specified message queue and returns 0 when successful, and returns 0. On failure, it returns -1 and sets **errno** as follows:

EAGAIN	The message queue is full, and the message queue is non-blocking.
EBADF	The **mqdes** argument is not a valid message queue descriptor.
EINVAL	The **msg_prio** argument is greater than or equal to MQ_PRIO_MAX.
EINTR	mq_send was interrupted by a signal.
EMSGSIZE	The message length specified by **msg_len** is more than the **mq_msgsize** attribute of the message queue.

CONFORMANCE

mq_send is implemented according to the POSIX 1003.1b (Real-Time) standard.

NOTE

The message priority has no relationship with the process scheduling priority. Message priority determines the order in which messages are added to (and removed from) a message queue. The process priority does, however come into the picture, when more than one processes is blocked on the same full message queue.

SEE ALSO

Library Calls - mq_receive(), mq_setattr(), mq_selective_receive()

NAME

`mq_setattr` – set the message queue attributes

SYNOPSIS

```
#include <mqueue.h>

int mq_setattr(mqdes, mqstat, omqstat)
mqd_t mqdes;
const struct mq_attr *mqstat;
struct mq_attr *omqstat;
```

DESCRIPTION

`mq_setattr` sets the attributes for the message queue specified by the **mqdes** argument to the argument specified by the **mqstat** argument.

If the **omqstat** argument is not `NULL`, `mq_setattr` retrieves the previous message queue attributes and stores them in the location referenced by this argument.

The structure `mq_attr` is defined as follows:

Type	Name	Description
`long`	`mq_flags`	Message queue flags
`long`	`mq_maxmsg`	Maximum number of messages
`long`	`mq_msgsize`	Maximum message size
`long`	`mq_curmsgs`	Number of messages currently queued

The meaning of the various members is described below.

The following member can be set at creation or set with the `mq_setattr` call at any time. It can be retrieved with the `mq_getattr` call.

`mq_flags` The message queue flags specify the blocking or non-blocking behavior of a message queue. If the `O_NONBLOCK` flag is set, the message queue is non-blocking. Otherwise, it is blocking. A blocking message queue causes the mq_send and mq_receive functions to block on full and empty message queues, respectively. A non-blocking message queue causes the mq_send and mq_receive functions to fail on full and empty message queues respectively.

The following members can only be set at the message queue creation, and can be retrieved with the mq_getattr call. These members cannot be set with the mq_setattr call. Values of these members in the **mqstat** argument are ignored by the mq_setattr call.

mq_maxmsg	This member specifies the maximum number of messages that can be allowed in the message queue. When the message queue contains mq_maxmsg messages, it is considered to be full. When mq_send is called on a full message queue that is blocking, mq_send blocks until space is available.
mq_msgsize	This member specifies the maximum size of each message in the message queue, in bytes. Attempts to send messages larger than mq_msgsixe will fail.

The following member provides status information about a message queue. This member cannot be set, and can only be retrieved with the mq_getattr call. The value of this of member in the **mqstat** argument is ignored by the mq_setattr call.

mq_curmsgs	Indicates the number of messages currently on the message queue.

DIAGNOSTICS

mq_setattr sets the attributes for the message queue when successful and returns 0. On failure, it returns -1 and sets **errno** as follows:

EBADF	The **mqdes** argument is an invalid message queue descriptor.

NOTE

Only the mq_flags member can be set with the mq_setattr call.

CONFORMANCE

`mq_setattr` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `mq_open()`, `mq_getattr()`

NAME

`mq_unlink` – remove a message queue

SYNOPSIS

```
#include <mqueue.h>

int mq_unlink(name)
const char *name;
```

DESCRIPTION

`mq_unlink` removes the message queue named by the string **`name`**. If there are one or more references to the message queue when it is unlinked, only the name is removed when `mq_unlink` returns. The message is removed only when all references to the message queue are closed.

Message queue names must be constructed with the standard pathname construction rules (see `mq_open`).

DIAGNOSTICS

`mq_unlink` removes the message queue when successful, and returns `0`. Otherwise it returns `-1`, and sets **`errno`** to one of the following values:

`EACCES`	The calling process does not have permission to unlink the message queue.
`ENAMETOOLONG`	The name of the message queue is too long, or a specific path component is too long.
`ENOENT`	The named message queue does not exist.
`ENOSYS`	The function `mq_unlink` is not supported.

CONFORMANCE

`mq_unlink` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `mq_open()`

NAME

mutex_create – create a user mutex

SYNOPSIS

```
#include <sem.h>

mutex_t mutex_create( )
```

DESCRIPTION

mutex_create initializes a new user mutex and returns a mutex descriptor for same.

A mutex is a mutual exclusion (hence the name) synchronization variable that can be used by threads to ensure that only one thread accesses a shared resource at a time. Between the time that a thread has returned from mutex_enter and the time it executes mutex_exit, no other thread will be allowed to successfully complete a mutex_enter call using that particular mutex. Threads that must ensure that they are the only threads accessing a particular shared resource should bracket the code accessing the resource with calls to mutex_enter and mutex_exit.

Of course, it may well be necessary for a thread to examine some condition associated with the shared resource and possibly block awaiting for some condition to be satisfied before proceeding with an operation on shared data. A condition variable can be used in conjunction with a mutex to allow the thread to atomically release the mutex and block awaiting a particular condition. Mutexes and condition variables are closely related in the way they are used. A mutex can be used without any condition variables, if nothing more than simple mutual exclusion is required. However, a condition variable must be used in conjunction with a mutex. Other uses of condition variables are unsafe and disallowed by the system.

NOTE

Using a mutex that has not been created with mutex_create, or which has been deleted with mutex_delete, leads to unpredictable results, up to and including segmentation violations and/or bus errors.

DIAGNOSTICS

`mutex_create` returns NULL if it fails to get the mutex. NULL is an invalid mutex descriptor. If the call fails, **errno** will contain one of the following:

`ENOSPC`	No more room for user mutexes.

SEE ALSO

Library Functions - `mutex_delete()`, `mutex_enter()`, `mutex_exit()`, `cv_wait()`, `cv_signal()`, `cv_broadcast()`, `csem_create()`, `csem_wait()`, `csem_signal()`

NAME

`mutex_delete` – delete a user mutex

SYNOPSIS

```
#include <sem.h>

int mutex_delete(md)
mutex_t md;
```

DESCRIPTION

`mutex_delete` frees up the system entry for the mutex identified by **`md`** and removes it from the list of active mutexes.

A mutex is a mutual exclusion (hence the name) synchronization variable that can be used by threads to ensure that only one thread accesses a shared resource at a time. Between the time that a thread has returned from `mutex_enter` and the time it executes `mutex_exit`, no other thread will be allowed to successfully complete a `mutex_enter` call.

Shared resources should be guarded with mutexes. If a thread holding a mutex must wait for some condition regarding the shared resource to be satisfied, then the thread should wait on a condition variable associated with that shared resource, using `cv_wait`.

DIAGNOSTICS

`mutex_delete` returns 0 if the mutex was deleted, or -1 if not.

If the call fails, **`errno`** will contain one of the following:

`EBUSY`	Some thread is waiting on the mutex.
`EINVAL`	Mutex **`md`** does not exist.

SEE ALSO

Library Functions - `mutex_create()`, `mutex_enter()`, `mutex_exit()`, `cv_wait()`, `cv_signal()`,

`cv_broadcast(), csem_create(),`
`csem_wait(), csem_signal()`

NAME

`mutex_enter` – acquire a mutex lock

SYNOPSIS

```
#include <time.h>
#include <st.h>
#include <sem.h>

int mutex_enter(m, timeout)
mutex_t m;
struct timeval *timeout;
```

DESCRIPTION

`mutex_enter` is used to acquire a mutual exclusion lock on a shared resource guarded by mutex **m**. A thread executing a `mutex_enter` on an unowned mutex will be granted ownership of the mutex. A thread executing a `mutex_enter` on an owned mutex will be blocked until the thread comes to the head of the queue of threads awaiting the mutex and the owner of the mutex releases ownership by executing a `mutex_exit` or a `cv_wait`. Threads waiting for a mutex are serviced in order of priority, highest-priority thread first.

The **`timeout`** argument specifies a maximum amount of time to await the mutex lock. If the argument is NULL, the thread blocks indefinitely. If the argument refers to a time of duration 0, the thread will not block at all.

NOTE

Using a mutex that has not been created with `mutex_create`, or which has been deleted with `mutex_delete`, leads to unpredictable results, up to and including segmentation violations and/or bus errors. Worse, the application may continue to run, but simply not synchronize its operations due to the faulty mutex.

DIAGNOSTICS

`mutex_enter` returns 0 when successful, or -1 upon error. If the call fails, **`errno`** will contain one of the following:

`EINVAL`	**m** does not refer to a valid mutex (This error may be returned if the application does not crash due to the invalidity of the mutex).
`EINTR`	The mutex wait was interrupted by a signal delivered to the thread.
`EINTR`	The mutex wait timed out without acquiring the mutex.

SEE ALSO

Library Functions - `mutex_exit()`, `mutex_create()`, `cv_wait()`, `cv_signal()`, `cv_broadcast()`, `csem_create()`, `csem_wait()`, `csem_signal()`

NAME

mutex_exit – release a mutex lock

SYNOPSIS

```
#include <time.h>

#include <st.h>

#include <sem.h>

int mutex_exit(m)
mutex_t m;
```

DESCRIPTION

mutex_exit is used by the owner of a mutex lock to release that lock. The mutex becomes unowned if and only if there are no threads currently awaiting ownership of the mutex; otherwise, the mutex ownership is granted to the thread that is of equal or greater priority to all other waiting threads and which has been waiting the longest for the mutex.

NOTE

Using a mutex that has not been created with mutex_create, or which has been deleted with mutex_delete, leads to unpredictable results, up to and including segmentation violations and/or bus errors. Worse, the application may continue to run, but simply not synchronize its operations due to the faulty mutex.

DIAGNOSTICS

mutex_exit returns 0 when successful, or -1 upon error. If the call fails, **errno** will contain one of the following:

EINVAL	**m** does not refer to a valid mutex. (This error may be returned if the application does not crash due to the invalidity of the mutex).
EPERM	The mutex is not owned by the thread.

SEE ALSO

Library Functions - `mutex_enter(), mutex_create(), cv_wait(), cv_signal(), cv_broadcast(), csem_create(), csem_wait(), csem_signal()`